SOFTWARE ARCHITECTURE

AUTHORS:
A. BIJLSMA
B.J. HEEREN
E.E. ROUBTSOVA
S. STUURMAN

FREE TECHNOLOGY ACADEMY

Software Architecture

A. Bijlsma
B.J. Heerendr.
E.E. Roubtsovair.
S. Stuurman

This version of the text is a preliminary version.
The final version will be published soon.

FREE TECHNOLOGY ACADEMY

Preface

Software has become a strategic societal resource in the last few decades. The emergence of Free Software, which has entered in major sectors of the ICT market, is drastically changing the economics of software development and usage. Free Software – sometimes also referred to as "Open Source" or "Libre Software" – can be used, studied, copied, modified and distributed freely. It offers the freedom to learn and to teach without engaging in dependencies on any single technology provider. These freedoms are considered a fundamental precondition for sustainable development and an inclusive information society.

Although there is a growing interest in free technologies (Free Software and Open Standards), still a limited number of people have sufficient knowledge and expertise in these fields. The FTA attempts to respond to this demand.

Introduction to the FTA
The Free Technology Academy (FTA) is a joint initiative from several educational institutes in various countries. It aims to contribute to a society that permits all users to study, participate and build upon existing knowledge without restrictions.

What does the FTA offer?
The Academy offers an online master level programme with course modules about Free Technologies. Learners can choose to enrol in an individual course or register for the whole programme. Tuition takes place online in the FTA virtual campus and is performed by teaching staff from the partner universities. Credits obtained in the FTA programme are recognised by these universities.

Who is behind the FTA?
The FTA was initiated in 2008 supported by the Life Long Learning Programme (LLP) of the European Commission, under the coordination of the Free Knowledge Institute and in partnership with three european universities: Open Universiteit Nederland (The Netherlands), Universitat Oberta de Catalunya (Spain) and University of Agder (Norway).

For who is the FTA?
The Free Technology Academy is specially oriented to IT professionals, educators, students and decision makers.

What about the licensing?
All learning materials used in and developed by the FTA are Open Educational Resources, published under copyleft free licenses that allow them to be freely used, modified and redistributed. Similarly, the software used in the FTA virtual campus is Free Software and is built upon an Open Standards framework.

Evolution of this book
The FTA has reused existing course materials from the Universitat Oberta de Catalunya and that had been developed together with LibreSoft staff from the Universidad Rey Juan Carlos. In 2008 this book was translated into English with the help of the SELF (Science, Education and Learning in Freedom) Project, supported by the European Commission's Sixth Framework Programme. In 2009, this material has been improved by the Free Technology Academy. Additionally the FTA has developed a study guide and learning activities which are available for learners enrolled in the FTA Campus.

Participation
Users of FTA learning materials are encouraged to provide feedback and make suggestions for improvement. A specific space for this feedback is set up on the FTA website. These inputs will be taken into account for next versions. Moreover, the FTA welcomes anyone to use and distribute this material as well as to make new versions and translations.

See for specific and updated information about the book, including translations and other formats: *http://ftacademy.org/materials/fsm/*. For more information and enrolment in the FTA online course programme, please visit the Academy's website: *http://ftacademy.org/*.

I sincerely hope this course book helps you in your personal learning process and helps you to help others in theirs. I look forward to see you in the free knowledge and free technology movements!

Happy learning!

Wouter Tebbens
President of the Free Knowledge Institute
Director of the Free technology Academy

Software architecture

Course book

Course team
dr. A. Bijlsma
dr. B.J. Heeren
dr. E.E. Roubtsova
ir. S. Stuurman

Software architecture

Course book

Contents

Contents **1**

I The activities of a software architect **3**

1 Introduction to software architecture **6**
Introduction 6
1.1 What is software architecture? 7
1.2 Topics in software architecture 10
1.3 The architect versus the engineer 16
Discussion questions 16

2 Requirements engineering **18**
Introduction 18
2.1 Important concepts 19
2.2 Use cases 22
2.3 Three types of requirements 24
2.4 Summary requirements engineering 26
2.5 Tactics 27
Discussion questions 27

3 Description and evaluation **30**
Introduction 30
3.1 Describing architectures 32
3.2 Architectural description languages 39
3.3 Evaluating architectures 43
Discussion questions 46

II Patterns in software architecture **47**

4 Architectural patterns **50**
Introduction 50
4.1 Architectural patterns or styles 51
4.2 Examples of architectural patterns 52
4.3 Architectural styles 65
4.4 Applying patterns: KWIC example 65
4.5 Choosing a style 68
Discussion questions 68

5 Patterns for enterprise applications **70**
Introduction 70
5.1 Enterprise applications 71
5.2 Patterns in the three layers 73

5.3 Concurrency problems . . . 85
Discussion questions . . . 87

6 Patterns for enterprise application integration 88
Introduction . . . 88
6.1 Enterprise application integration . . . 89
6.2 Loose and tight coupling . . . 90
6.3 Integration patterns . . . 92
6.4 Conclusion . . . 102
Discussion questions . . . 103

III Components and contracts 105

7 Component software 108
Introduction . . . 108
7.1 Custom-made, standard and component software . . . 109
7.2 The nature of components . . . 109
7.3 Components and objects . . . 110
7.4 Reuse . . . 111
7.5 UML and components . . . 111
7.6 Concepts associated with components . . . 115
7.7 Connecting components . . . 117
7.8 Future and current approaches . . . 118
Discussion questions . . . 122

8 Design by contract 124
Introduction . . . 124
8.1 Contracts . . . 125
8.2 Problems with contracts . . . 126
8.3 Polymorphism . . . 130
8.4 Inheritance . . . 132
8.5 Delegation . . . 135
8.6 Techniques for specification . . . 135
Discussion questions . . . 137

9 Service oriented architectures 138
Introduction . . . 138
9.1 Aspects of services . . . 139
9.2 Standards and technologies . . . 141
9.3 Coordination of services . . . 142
9.4 Orchestration . . . 143
9.5 Choreography . . . 144
9.6 Security and reliability . . . 144
9.7 Platforms for service-oriented architectures . . . 145
9.8 RESTful webservices . . . 146
Discussion questions . . . 146

Bibliography 148

Block I

The activities of a software architect

Module 1

Introduction to software architecture

INTRODUCTION

Software architecture is the subject of this course. In this introduction, we will explain what software architecture is. You will find out why we need software architecture and you will see aspects of scientific research that is done on software architecture. You will be introduced to some of the topics within the field of software architecture that will be covered in this course: software architecture as a solution balancing the concerns of different stakeholders, quality assurance, methods to describe and evaluate architectures, the influence of architecture on reuse, and the life cycle of a system and its architecture. This learning unit concludes with a comparison between the professions of software architect and software engineer.

All aspects of software architecture that are introduced in this module will be treated in more detail in individual modules.

LEARNING OBJECTIVES

After having studied this module you should be able to:

- describe the place of software architecture in the life cycle,
- explain the need for an architecture,
- describe the responsibilities of a software architect,
- explain the relationship between stakeholders and the architecture for a system,
- describe the role of requirements in software architecture,
- explain the role of compromise in creating an architecture,
- explain the relationship between architecture and reuse.

1.1 What is software architecture?

1.1.1 A definition

DEFINITION 1.1 The IEEE standard 1471 defines *software architecture* as 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.

An architecture embodies information about components and their interaction, but omits information about components that does not pertain to their interaction. Thus, an architecture is foremost an abstraction of a system that suppresses details of components that do not affect the use, the relations and interactions of components. Private details of components, details that have to do solely with internal implementation and are not externally visible, are not architectural. In short, an architecture determines how components interact, not how they are implemented.

Every system has an architecture, but it does not follow that the architecture is known to anyone. Perhaps the designers of the system are long gone, perhaps documentation was never produced, and perhaps the source code has been lost.

This shows that an architecture is not the same as a description of an architecture. While most learning units concern different aspects of architectures, one of the learning units is explicitly dedicated to the description of architectures.

1.1.2 Architecture in the life cycle

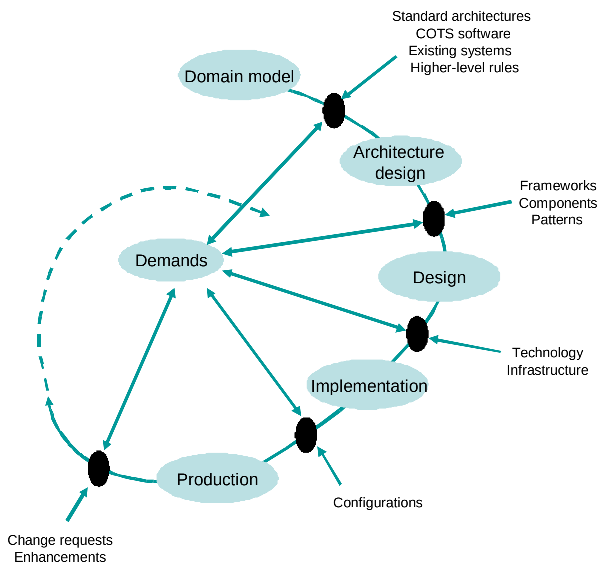

Figure 1.1: System life cycle

Figure 1.1 shows the life cycle of a system, and the place of architecture in the life cycle. Software architecture links requirements analysis with realization and it may be explored and defined incrementally. Its existence allows to predict quality aspects before realization. The architecture must be stable before major development starts.

An architecture can help evolutionary development, when evolution is one of the requirements. A software architecture may be seen as a blueprint and guideline for realization.

DEFINITION 1.2 *Architectural conformance* is the extent to which the architecture is actually used.

Conformance may be enforced by management. For optimal conformance, an architecture should be well documented. An architecture should also state clear principles, for instance, "no database access from the servlet layer". Architectural conformance should also be maintained over time.

Architectural decay is the opposite of architectural conformance over time: it is often the case that software drifts from the original architecture through maintenance and change operations.

1.1.3 Why do we need software architecture?

Applications are becoming larger, more integrated, and are implemented using a wide variety of technologies. The various technologies and disciplines need to be orchestrated to ensure product quality. Quality attributes like reliability or usability cannot be analyzed at the code level, but they can be analyzed at the software architectural level.

Software architecture has several functions, which we describe below.

Software architecture as a means for communication

In the first place, we need an architecture as a means for communication among stakeholders.

DEFINITION 1.3 A *stakeholder* is anyone with a legitimate interest in the construction of a software system. Stakeholders include the owner, the end users, the developers, project management, and the maintainers, amongst others. Stakeholders are representatives from all stages of development, usage and support.

Owners, users, developers as well as maintainers, all are considered stakeholders. A software architecture represents a common high-level abstraction of a system that can be used by all of the system's stakeholders as a basis for creating mutual understanding, forming consensus, and communicating with each other [6].

In open source software development, the following stakeholders may be discerned: users, contributers and committers (such as developers), owners (sometimes, a project is owned by a company, while the resulting software is open source) and providers (for instance businesses that contribute money to a project, or allow developers to work during work hours on a project) [41].

Software architecture as a representation of early design decisions

A second function of software architecture is that it is a representation of *early design decisions*. A software architecture is the documentation of the earliest design decisions about a system, and these early decisions carry weight far out of proportion to their individual gravity with respect to the system's remaining development, its deployment, and its maintenance life. It is also the earliest point at which the system to be built can be analyzed.

Software architecture as a basis for work-breakdown structure

Software architecture does not only prescribe the structure of the system being developed. That structure also becomes engraved in the structure of the development project. Because the architecture includes the highest level decomposition of the system, it is typically used as the basis for the *work-breakdown* structure. This dictates units of planning, scheduling, and effectively freezes the architecture. Development groups typically resent relinquishing responsibilities, which means that it is very difficult to change the architecture once these units have begun their work.

Software architecture as a means to evaluate quality attributes

It is possible to decide that the appropriate architectural choices have been made (i.e. that the system will exhibit its required quality attributes) without waiting until the system is developed and deployed, because software architecture allows to predict system qualities. This is further explained in the learning unit dedicated to architecture evaluation.

Software architecture as a unit of reuse

Another function of software architecture is that it is a transferable abstraction of a system. A software architecture constitutes a relatively small, intellectually graspable model of the structure of a system and the way in which its components work together. This model is transferable across systems. In particular, it can be applied to other systems exhibiting similar requirements and can promote large-scale reuse.

An example of how software architecture promotes large scale reuse is the construction of product lines. A *software product line* is a family of software systems that shares a common architecture. The product line approach is a way to gain quality and save labor.

Another example of how software architecture promotes large scale reuse is component-based development. Software architecture complements component-based and services-based development.

Many software engineering methods have focused on programming as the prime activity, with the progress measured in lines of code. Architecture-based development often focuses on composing or assembling components that are likely to have been developed separately, even independently, from each other.

1.1.4 Software architecture as a discipline

Software architecture has become a discipline with the article of Mary Shaw in 1989: "Larger Scale Systems Require Higher Level Abstractions" [46]. She showed that organizing systems on the subsystem and module level is different from organizing code.

The field of software architecture has taken inspiration from other engineering domains such as architecture and electronics. The concepts of stakeholders and concerns, analysis and validation, styles and views, standardization and reuse, best practices and certification are all well-known in these domains. However, software is different from products in all other engineering disciplines. Rather than delivering a final product, delivery of software means delivering blueprints for products. Computers can be seen as fully automatic factories that accept such blueprints and instantiate them.

The important consequence is that plans can be parameterized, applied recursively, scaled and instantiated any number of times. There are no reproduction costs. Unfortunately, invisible complexity may lead to unreasonable assumptions, for instance about flexibility.

1.1.5 Scope and focus of architecture

Scope and focus are two dimensions of software architectures.

DEFINITION 1.4 The *scope* of a software architecture is the range of applications to which it pertains.

An architecture describes at least a single system or product, such as a specific version of OpenOffice Writer. A more general scope is that of a system family. An example of a system family is the OpenOffice software suit. This family consists of several products, for word processing, spreadsheets, presentations, graphics, databases and more, but all built along similar architectural lines [37]. The scope may be that of an organization as a whole, of a domain, or the scope may be generic.

An example of a domain architecture is a compiler. A compiler generally consists of several basic elements (the front end, back end, symbol table and such), that behave in a well known way and are interconnected in a regular fashion. Someone designing a compiler would not start from scratch, but would begin with this basic domain architecture in mind when defining the software architecture of a new compiler.

Another aspect in which architectures differ is their *focus*:

– An *application architecture* is a blueprint for individual applications, their interactions, and their relationships to the business processes of the organization. It is built on top of the IT architecture.
– *Information architecture* is concerned with logical and physical data assets and data management resources.
– An *IT architecture* defines the hardware and software building blocks that make up the overall information system of the organization. The business architecture is mapped to the IT architecture. The purpose of an IT architecture is to enable the organization to manage its IT investment in a way that meets its needs. It includes hardware and software infrastructure including database and middleware technologies.
– The *business architecture* defines the business strategy, governance, organization, and key business processes within an organization. The field of business process reengineering (BPR) focuses on the analysis and the design of business processes, not necessarily represented in an IT system.

1.2 Topics in software architecture

Topics within the field of software architecture are:

– **Definition of the solution structure**. The solution structure is defined using concepts such as components and connectors, contracts, frameworks, and services.
– **Quality assurance**, in relation to stakeholders and their concerns. A rationale explains how a certain decision is related to concerns of stakeholders. You can use analysis, validation or assessment.
– **Describing architectures**. Concepts used are viewpoints, models and views. You can use languages, notations and visualization.
– **Architecture and reuse**. Concepts are reusable architectures, architectural styles and patterns. Specification and connection of components, standardization, components, product lines, redesign of legacy systems and service-oriented architectures all are related to reuse.
– **Architecture in the life cycle** of a system: there is need for a methodology for maintenance and evolution of the architecture.

We will introduce these topics in this learning unit; you will read about them in more detail in following learning units.

1.2.1 Stakeholders and their concerns

Examples of *stakeholders* are owners, end users, developers, the developer's organization, and those who maintain the system. An architecture must balance the concerns of the different stakeholders.

All stakeholders have their own concerns. Users, for example, want a certain behavior at runtime, performing well on a particular piece of hardware. Developers may want to gain experience in a certain programming language, or in a modeling technique.

DEFINITION 1.5 A *concern* is an interest related to the development, operation, use or evolution of a system. A concern may be related to functionality, quality areas, costs, technology, and so forth.

Examples of concerns are providing a certain behavior at runtime, performing well on a particular piece of hardware, being easy to customize, achieving a short time to release, or gainfully employing programmers who have a particular specialty.

Concerns are very important: it is costly or impossible to repair or ignore them. Concerns may be contradictory. Speed, flexibility or reliability for instance, may ask for different solutions. Therefore it is important to prioritize.

The traden offs between performance and security, between maintainability and reliability, between the cost of initial development and the cost of future developments, are all manifested in the architecture. The system's behavior on these quality issues is the result of structural trade-offs made by the developers, and is traditionally undocumented!

Architects must identify and actively engage the stakeholders to solicit their needs and expectations. Without such active engagement, the stakeholders will, at some point, explain to the architects why a proposed architecture is unacceptable, thus delaying the project. Early engagement allows architects to understand the constraints of the task, to manage expectations, and to negotiate priorities.

The architect must understand the nature, source, and priority of these constraints. This places the architect in a better position to make trade-offs when conflicts among competing constraints arise, as they inevitably will.

The architect needs more than just technical skills. Stakeholders will have to be informed on a continuing basis why priorities have been chosen, and why some expectations cannot be met. Diplomacy, negotiation, and communications skills are essential.

1.2.2 The QUINT/ISO 9126 quality framework

DEFINITION 1.6 A *quality framework* establishes a terminology for quality attributes (as a common language for negotiation with and among stakeholders) and establishes measures for the attributes.

Figure 1.2 presents the properties that are used as parameters of quality. Quality must be considered at all phases of design, implementation, and deployment, but different qualities manifest themselves differently during these phases. For instance, many aspects of usability are not architectural: making the user interface clear and easy to use is primarily a matter of getting the details correct (radio button or check box?). These details matter tremendously to the end user, but they are not always architectural.

The quality properties that are important in the requirements are assessed from the description of an architecture. When analysis of the description shows that the desired properties will not be fulfilled, improvements to the architecture are needed. Properties can not always be automatically analyzed, or even analyzed at all or at this early

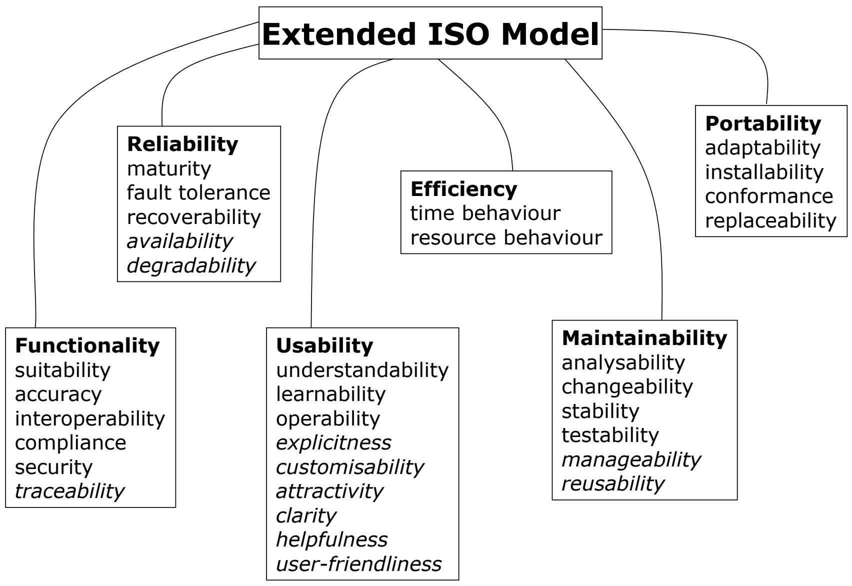

Figure 1.2: Parameters of quality

stage of system design. Therefore, reviews of experts are also an important method for quality assurance.

Techniques for *quality assurance* are: applying specific measures to design, technology, process and so on, building architectural prototypes, reviews by experts, assessment methods and methods for automatic analysis (which require suitable models and descriptions).

1.2.3 Describing architectures

Quality assessment of architectural decisions requires descriptions of architectures. An important concept in architecture description is the notion of *viewpoints*. Different stakeholders require descriptions of the architecture from different viewpoints: developers, maintainers, end-users, project managers, service engineers, auditors or other architects are interested in different aspects of the architecture. Several reference models for architectural descriptions using different viewpoints exist, such as the 4+1 model of Kruchten [27], the viewtypes of Clements and others [12] or the viewpoints and perspectives of Rozanski and Woods [43].

Architectures are described using several languages. Natural languages and the Unified Modeling Language (UML) [35] are often used, as are pictures without a formal basis. More formal notations are also used: Module Interconnection Languages (MILs) and Architecture Description Languages (ADLs). These are names for classes of languages. Some of these languages are treated in the learning unit on describing and evaluating architectures.

An architecture can be characterized as a set of *models*. We build models of complex systems because we cannot comprehend such a system in its entirety: a model is a simplification of reality. We build models to understand the system we are building.

Models can be used for visualization, specification, as a template for construction, or for documenting decisions. Every model can be expressed at different levels of precision.

A *view* is a projection of a model, omitting entities that are irrelevant from a certain perspective or vantage point. Each viewpoint may define one or more views. No single model suffices. Systems are best approached through a small set of nearly independent models with multiple views.

- Views belonging to the *functional viewpoint* describe the system's runtime functional elements and their responsibilities, interfaces, and primary interactions. The functional view of a system defines the architectural elements that deliver the system's functionality. These views document the system's functional structure-including the key functional elements, their responsibilities, the interfaces they expose, and the interactions between them. Diagrams used in these views model elements, connectors, interfaces, responsibilities and interactions, for instance using UML component diagrams.
- Views belonging to the *information viewpoint* describe the way that the architecture stores, manipulates, manages, and distributes information. This viewpoint concerns both the information structure and the information flow. Diagrams for these views are for instance DFD's (Data Flow Diagrams), UML Class diagrams, ER (entity-relation) diagrams, and so on.
- Views belonging to the *concurrency viewpoint* describe the concurrency structure of the system, mapping functional elements to concurrency units to clearly identify the parts of the system that can execute concurrently, and shows how this is coordinated and controlled. This viewpoint is concerned with the system's concurrency and state-related structure and constraints. Diagrams which can be used for these views are for instance UML state diagrams or UML component diagrams.
- Views belonging to the *development viewpoint* describe the architecture that supports the software development process. Aspects are module organization, standardization of design and testing, instrumentation, code structure, dependencies, configuration management. Diagrams for these views are for instance UML component diagrams with packages.
- Views belong to the *deployment viewpoint* describe the environment into which the system will be deployed, including the dependencies the system has on its runtime environment. Aspects in this viewpoint are specifications of required hardware, of required software, or network requirements. Diagrams for these views are UML deployment diagrams.
- Views belonging to the *operational viewpoint* describe how the system will be operated, administered, and supported when it is running in its production environment. Aspects in this viewpoint are installation and upgrade, functional migration, data migration, monitoring and control, backup, configuration management and so on.

Depending on the nature of the system, some views may be more important than others. In data-intensive systems, views addressing the information viewpoint will dominate. In systems which will have many short release cycles, the deployment viewpoint and the views belonging to it will be important. In real time systems, the concurrency viewpoint will be important.

The choice of what viewpoints to use has a profound influence on how a problem is attacked and a solution is shaped. The viewpoints you choose greatly affect your world view. If you build a system through the eyes of a database developer, you will likely focus on entity-relationship models. If you build a system through the eyes of a structured analyst, you will likely end up with models that are algorithm-centric, with data flowing from process to process. If you build a system through the eyes of an object-oriented developer, you will end up with a system with an architecture centered around classes and their interactions. Each world view leads to a different kind of system, with different costs and benefits.

1.2.4 Architecture and reuse

Architectures are reusable artifacts. Reuse comes in different forms. Architectural patterns and styles may be reused. Frameworks with infrastructural support are another way of reuse. An example of a framework is the Struts framework for web applications using the Model-view-controller pattern [4].
To enable reuse, architecture standards are needed, with clear “component” roles. These standards may be aimed at a specific domain, or at a specific technology.

Product lines

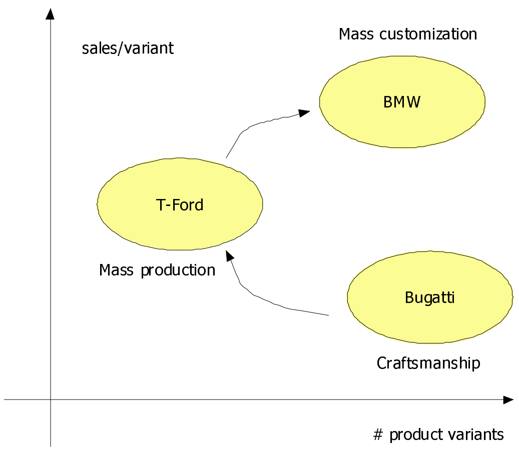

Figure 1.3: Toward product lines within the automotive industry

Product lines form an example of reuse-driven development. Figure 1.3 shows the number of product variants from left to right, and the number of sales per variant from below to above. It shows how the production of individual cars using traditional craftsmanship (Bugatti) has evolved to mass production of a single model (T-Ford), and later to mass production and mass customization, which makes it possible to produce multiple models in great quantities. Mass customization in the automotive industry is what product lines are in the software industry.

DEFINITION 1.7 A *software product line* (SPL) is a set of software-intensive systems that share a common, managed set of functional modules and non-functional features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [11].

In other words, a software product line is a family of similar systems. The commonality of the systems is called the *commonality*; the variations are called the *variability*. Requirements for products within a product line are modeled in a feature model (where feature stands for a requirement from a user’s perspective), which makes it possible to discern common features for all products within the product line, and features that vary among different products.
Common components or a framework offer the common features; application-specific extensions offer the variability. Domain engineering addresses the systematic creation

of domain models and architectures, leading to common components or frameworks. Application engineering uses the models and architectures (and the common components and frameworks) to build systems.

In the words of the Carnegie Mellon Software Engineering Institute (SEI) [11]:

> **What is a Software Product Line?**
> A software product line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.
>
> **Why are Software Product Lines Important?**
> Software product lines are rapidly emerging as a viable and important software development paradigm allowing companies to realize order-of-magnitude improvements in time to market, cost, productivity, quality. Software product line engineering can also enable rapid market entry and flexible response, and provide a capability for mass customization.

Reverse engineering

Development often involves legacy systems that need to be reused, and sometimes reorganized.

DEFINITION 1.8 A *legacy system* is an existing computer system or application program which continues to be used because the user (typically an organization) does not want to replace or redesign it.

When we deal with a legacy system we rarely have up-to-date documentation. The architecture models often need to be recovered, the data types transformed and the source code refactored.

DEFINITION 1.9 The process of recovering the architecture of a legacy system is called *reverse engineering*.

Reverse engineering is a process of examination; the software system under consideration is not modified. Doing the latter would make it *re-engineering*.

Refactoring means modifying source code without changing its external behavior. Refactoring does not fix bugs or add new functionality. The goal of refactoring is to improve the understandability of the code, change its structure and design, remove dead code, or make it easier to maintain the code in the future.

An example of a trivial refactoring is changing a variable name into one conveying more information, such as from a single character *'i'* to *'interestRate'*. A more complex refactoring is to turn code within an if-block into a subroutine.

Reverse engineering is based on either human cognition or on technical approaches.

- Cognitive strategies are based on how humans understand software. A human might take a top-down approach to understanding software (start at the highest level of abstraction and recursively fill in understanding of the subparts), a bottom-up approach (start by understanding the lowest-level components and how these work together), a model-based approach (if the investigator already has a mental model of how the system works and tries to deepen the understanding of certain areas), or an opportunistic approach (some combination of these approaches).

– Technical approaches center on extracting information from the system's artifacts, including source code, comments, user documentation, executable models, system descriptions, and so forth. Successful techniques of reverse engineering are extracting cross-reference information using compiler technology, data flow analysis, profiling to determine execution behavior, natural language analysis, and architectural-level pattern matching on the source code.

1.3 The architect versus the engineer

Software architecture is a separate domain that requires professionals: software architects. Compared with the work of an engineer, who analyzes the precise requirements of a system, develops a detailed solution, and ensures proper implementation and working of a system or a part of a system, a software architect creates a vision (an architecture), manages stakeholders' expectations and maintains the vision.

DISCUSSION QUESTIONS

1. Discuss the role of software architecture in open source development.
2. Do all stakeholder concerns carry equal weight? If not, what criteria exist for arbitrating among them?
3. Where does the analogy between software architecture and building architecture break down?
4. Discuss the role of software architecture with respect to agile methods such as Extreme Programming.
5. Figure 1.4 shows the overview of the software architecture of the OpenOffice suit architecture [37], as it is shown in the documentation. Discuss to which viewpoint(s) this view belongs.

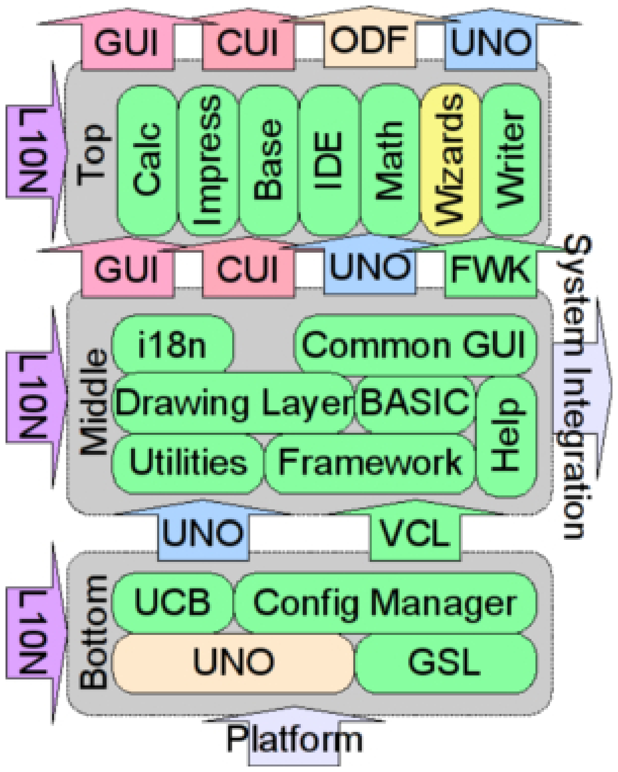

Figure 1.4: Architecture Overview

Module 2

Requirements engineering

INTRODUCTION

Choosing the software architecture for a system is the first step toward a system that fulfills the requirements. Software requirements, therefore, are important to the field of software architecture. Software requirements engineering is needed to understand and define what problem needs to be solved. We need to discover, understand, formulate, analyze and agree on what problem should be solved, why such a problem need to be solved, and who should be involved in the responsibility of solving that problem [28].

In this module, we will discuss the concepts of stakeholder and concern, which are both important with respect to requirements. We will discuss the different types of requirements, and show a way to categorize them. We will show you different ways to elicit requirements from stakeholders, ways to specify them, and ways to validate them. Use cases are often used for specification of software requirements; we will discuss them in more detail. We will also introduce tactics as a strategy to meet quality requirements.

LEARNING OBJECTIVES

After having studied this module you should be able to:

- explain the notions of stakeholder and concerns,
- summarize the different types of requirements,
- state the steps to be taken in the requirements engineering process,
- describe how use cases can be used for specifying functional requirements,
- describe why concerns have to be prioritized,
- describe the relationship between tactics and requirements.

2.1 Important concepts

2.1.1 Stakeholders and concerns

The notions of stakeholder and concern are defined in definitions 1.3 and 1.5.

Examples of stakeholders are:

- Acquirers are those who decide which system to use,
- Assessors are those who check whether a given system conforms to needs or constraints,
- Communicators are responsible for training and documentation
- Developers develop the system,
- Contributers develop or write documentation,
- Committers take decisions in the development process,
- Maintainers fix bugs and evolve the system,
- Suppliers provide components,
- Supporters provide help to users,
- System administrators keep the system running, administer users, and configure the system,
- Testers test (parts of) the system,
- Users use it.

In many cases, it is useful to split some of those groups. Users of a content management system for instance, may be divided in users who enter content, and users who only consume content.

The concerns of different stakeholders are often mutually incompatible. Examples of such incompatible concerns are performance versus security or modifiability versus low cost. Concerns of the developer are not the same as those of the user or the acquirer. The developer might be interested in using new techniques and experimenting, while the user may want a product which is stable, and the acquirer wants a product which can easily be adapted or configurated. This means that the architecture of a system is usually a compromise or trade-off among incompatible concerns. It is important to document and explicitly prioritize concerns. An architecture is the earliest artifact that allows the priorities among competing concerns to be analyzed.

Standard architectures and architectural patterns have well known effects on the quality attributes of a system. This makes it possible to reason about the effect of the chosen architecture on the qualities of the system that are requested by the different stakeholders. Therefore, architectural decisions can be analyzed at an early stage (before design starts), to determine their impact on the quality attributes of the system. It is worth doing this analysis because it is virtually impossible to change the architecture at a later stage of development.

2.1.2 Software requirements

According to the IEEE Standard Glossary of Software Engineering Terminology [49], the definition of a software requirement is:

DEFINITION 2.1 A *software requirement* is a condition or capacity needed by a user to solve a problem or achieve an objective.

Requirements come in three types: functional, non-functional and constraints:

- *Functional requirements* present what the system should do. The rules of an online game are examples of functional requirements.

- *Non-functional requirements* specify with what quality the system should work. A possible non-functional requirement for an online game is that the game should provide an interface that is easy to understand, or that the response on an action of the user should be given within less than a certain maximum time.
- *Constraints* show the limits within which the system should be realized. For an online game, a constraint could be that it should work with both Firefox and Internet Explorer, without the need for a plugin.

We will discuss these different types of requirements further on in this module.

2.1.3 Requirements engineering

Requirements engineering is a cyclic process involving [28]:
- Domain understanding and elicitation;
- Evaluation and negotiation;
- Specification and documentation;
- Quality assurance: validation and verification.

A technique for requirements engineering as a whole is the KAOS method [28]. The *KAOS method* offers techniques to model a systems requirement in the form of goals, conceptual objects, agents, operations, behavior, and the relations between those entities.

DEFINITION 2.2 *Domain understanding* means: acquiring a good understanding of the domain in which the problem is rooted, and what the roots of the problem are. [28].

The following aspects are important for domain understanding:
- The organization within which the current system (which may already be supported by software or not) takes place (if applicable): its structure, objective and so on;
- The scope of the current system;
- The set of stakeholders to be involved;
- The strength and weaknesses of the current system.

DEFINITION 2.3 *Requirements elicitation* means: the activity of discovering candidate requirements and assumptions that will shape the system-to-be, based on the weaknesses of the current system as they emerge from domain understanding. [28].

Techniques for requirements elicitation are:
- Asking: interview, (structured) brainstorm, questionnaire,
- Scenario-based analysis: ‘think aloud’, use case analysis, storyboards,
- Ethnography: active observation,
- Form and document analysis,
- Knowledge reuse: reuse requirements from similar systems,
- Start from an existing system,
- Prototyping and mock-ups,
- Following your own insight.

None of these techniques guarantees ‘correct’ requirements. Actually, it is impossible to formulate what correct requirements mean. It is generally advisable to use more than one technique.

In open source projects, requirements are often gathered ‘on-the-fly’ by releasing a working product, establish a user community, and involving that user community by offering a means to ask for new features, report bugs, and comment on the offered software product.

DEFINITION 2.4 *Requirements evaluation* means: making informed decisions about issues raised during the elicitation process. [28].

Negotiation may be required in order to reach a consensus.

- Conflicting concerns must be identified and resolved.
- The risks associated with the system must be assessed and resolved.
- Alternative options must be compared.
- Requirements must be prioritized.

DEFINITION 2.5 *Requirements specification* means: rigorous modeling of requirements, to provide formal definitions for various aspects of the system ([30]).

A requirements specification document should be:

- as precise as possible: it is the starting point for architecture and design,
- as readable as possible: it should be understandable for the user.

Preferred properties of a requirements specification are that the specification is correct, unambiguous, complete, consistent, ranked for importance, verifiable, modifiable and traceable.

Among the techniques for requirements specification are Entity-Relationship (E-R) modeling, the Structured Analysis and Design Technique (SADT), Finite State Machines, use cases and UML.

- *E-R modeling* is a semantic data modeling technique, developed as an extension to the relational database model (to compensate for its absence of typing and inadequate modeling of relations), not unlike the UML class diagram.
- *SADT* has been developed in the late 1970s by Ross [42]. It is based on a dataflow model that views the system as a set of interacting activities. As notation it uses rectangles representing system activity, with four arrows. An arrow from the left signifies input, an arrow to the right signifies output, an arrow from above signifies control or database, and an arrow from below signifies a mechanism or algorithm.
- *Finite state machines* show states and transitions, with guards and actions. A finite state machine models the different states the system can be in, where a state is characterized by the actions that are enabled. Finite state machine modeling is part of UML as state diagrams.
- *UML* incorporates class models, state diagrams and use cases — but not data flow (data flow diagrams mix static and dynamic information, and are replaced by activity and collaboration diagrams). Use cases will be treated in more detail below.

DEFINITION 2.6 *Requirements validation* is concerned with checking the requirements document for consistency, completeness and accuracy ([26]).

Requirements should be validated with stakeholders in order to pinpoint inadequacies with respect to actual needs. [28]

Requirements verification is something else: a mathematical analysis, possibly automated, of formal specifications for consistency. Verification only checks consistency; completeness or accuracy cannot be checked with mathematical analysis. You need user interaction to validate requirements: the requirements document may not reflect the real requirements.

Among the techniques for requirements validation are:

- Reviews (walkthroughs, reading, checklists, discussion),
- Prototyping (the process of constructing and evaluating working models of a system to learn about certain aspects of the required system and/or its potential solution.).
- Animation (the ability to depict graphically the dynamic behavior of a system by interactively walking through specification fragments to follow some scenario [10]).

2.2 Use cases

Use cases form a technique for specifying functional requirements: use cases are helpful to specify what the system should do. A use case captures a contract with the stakeholders of a system about its behavior. A use case describes the system's behavior under various conditions, as the system responds to a request from one of the stakeholders, called the primary actor. Use cases are fundamentally a description of usage scenarios in textual form.

An example of a use case is the following:

```
Use case:
   Withdraw cash from ATM
Level:
   User goal
Primary actor:
   Account holder
Description:
   Customer enters ATM card
   ATM
      reads the bank ID,
         account number,
         encrypted PIN from the card,
      validates the bank ID and account number
      with the main banking system
   Customer enters PIN.
   ATM validates it against the encrypted PIN from the card.
   Customer selects 'Withdraw cash' and withdrawal amount
   ATM notifies main banking system of customer account and amount,
   and receives bank acknowledgment
   ATM delivers cash, card, and a receipt
   ATM logs the transaction
```

As you can see, the basic format for a use case is the following:

```
Use case:
   <use case goal>
Level:
   <one of: summary level, user-goal level, subfunction>
Primary actor:
   <a role name for the actor who initiates the use case>
Description:
   <the steps of the main success scenario
   from trigger to goal delivery and any cleanup after>
```

Several use cases may be combined in a *use case diagram*. Figure 2.1 shows an example. The line drawn between an actor and a use case is an association: the participation of

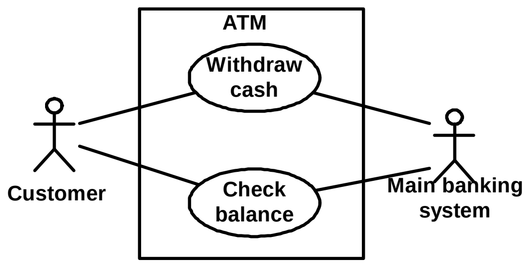

Figure 2.1: Use cases for an ATM

an actor in a use case. Instances of the actor and instances of the use case communicate with each other. This is the only relationship between actors and use cases.

An actor can be thought of as a role. In Figure 2.1 we see two actors: the customer and the main banking system. Both actors are associated with two use cases: withdraw cash and check balance.

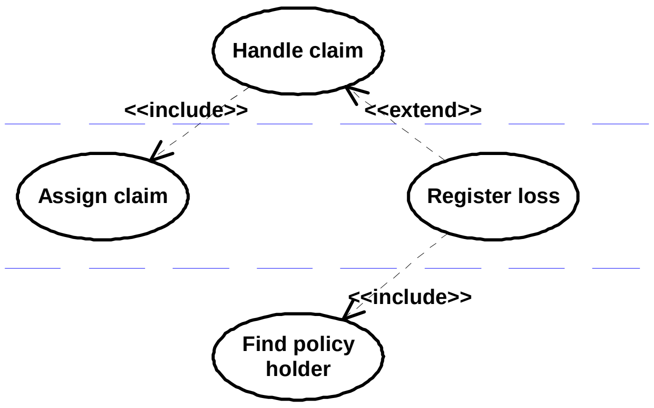

Figure 2.2: Different levels in use cases

Use cases can be described at different levels:

- A use case of *summary level* involves multiple user goals, shows the context of user goals, the life-cycle of user goals, or a table-of-contents. An example is a use case such as *Handle an insurance claim* (see Figure 2.2).
- A use case of *user goal level* shows the goal a primary actor has in trying to get work done. An example of a use case on the use level is *Register a loss*.
- A *subfunction* is a step needed to carry out a user goal. An example is *Find policy holder's file*.

In Figure 2.2 we see three annotations on the relationships between the different levels. These are:

- An *extend* relationship from a use case A to a use case B indicates that an instance of use case B may be augmented (subject to specific conditions specified in the extension) by the behavior specified by A. The behavior is inserted at the location defined by the extension point in B, which is referenced by the extend relationship.
- A *generalization* from a use case C to a use case D indicates that C is a specialization of D.

- An *include* relationship from a use case E to a use case F indicates that an instance of the use case E will also contain the behavior as specified by F. The behavior is included at the location which defined in E. (see [33]).

Some advice on how to write use cases:

- A use case is a prose essay. Make the use cases easy to read using active voice, present tense, describing an actor successfully achieving a goal.
- Include sub-use cases where appropriate.
- Do not assume or describe specifics of the user interface.
- An actor is not the same as an organizational role: an actor is a person, organization, or external system that plays a role in one or more interactions with the system.
- Use UML use case diagrams to visualize relations between actors and use cases or among use cases. Use text to specify use cases themselves!
- It is hard, and important, to keep track of the various use cases.

2.3 Three types of requirements

2.3.1 Functional requirements

Pitfalls with respect to functional requirements are [29]:

- Having an undefined or inconsistent system boundary. The system boundary defines the scope of the system: what does and what does not belong to the system. The system boundary, therefore, determines which problems the system should solve (and which problems belong to the world outside the system). Within the system, the boundary between the responsibilities of the software and of actors within the system should be clear.
- Describing use cases from the point of view of the system instead of describing them from the actor: the correct point of view is that from the actor.
- Using inconsistent actor names: actors names should be consistent throughout.
- Creating spiderwebs of actor-to-use case relations: relations between actors and use cases should be clear.
- Writing long, excessive, or confusing use case specifications, incomprehensible to the stakeholders: use case descriptions should be clearly understandable (and understood) by the stakeholders.

Beware of:

- A 'shopping cart' mentality. Stakeholders often have the tendency to treat requirements as items that can be put into a shopping cart. You should always make clear that every requirement comes at a price.
- The 'all requirements are equal' fallacy: architectural requirements must be prioritized to indicate to the architect, or anyone else, which requirements are most important to the finished system. No design trade-offs can be made if all requirements are assigned the same priority.
- Stakeholders who will not read use case descriptions because they find them too technical or too complicated. It is important to assure that your stakeholders understand the value of taking time to understand the descriptions.

2.3.2 Quality requirements

Quality requirements are the main category of non-functional requirements. Quality requirements are important parameters for defining or assessing an architecture. For example, the architecture of a safety critical system will differ from the architecture of a computer game. Quality requirements may be specified using a *software quality* *model*. A software quality model serves as a discussion framework, a scheme for users and developers to talk about different kinds of quality, to prioritize different kinds of quality, and to check the completeness of quality requirements.

ISO 9126 quality model

ISO 9126 [25] is the international standard quality model. The model classifies software quality in a structured set of factors:

- Functionality
- Reliability
- Usability
- Efficiency
- Maintainability
- Portability

Every factor is divided into a set of sub-characteristics. For example,

```
Maintainability =
  {Stability, Analyzability, Changeability, Testability}
```

Every sub-characteristic is divided into a set of attributes, an entity which can be measured or verified. The attributes are not defined in the standard quality mode.

Quint2 quality model

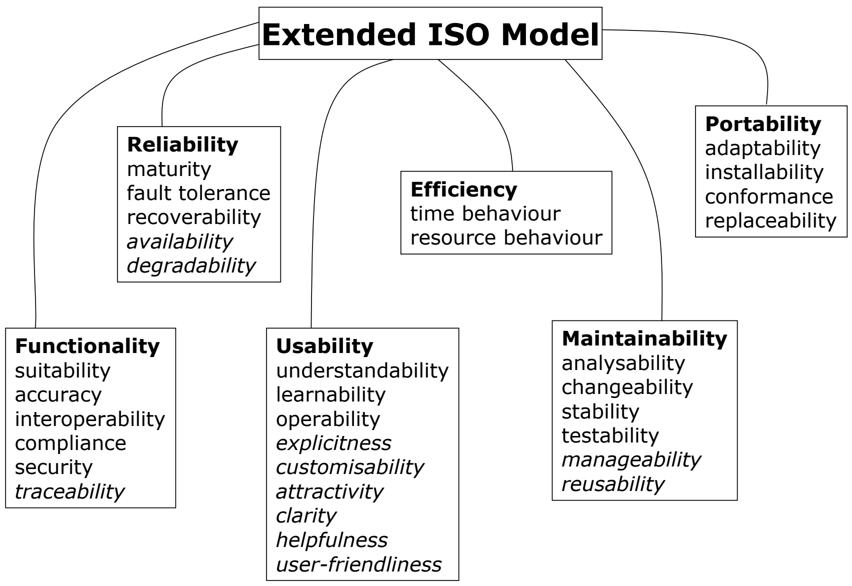

Figure 2.3: Parameters of quality

Quint2 (Quality in Information Technology) is a Dutch framework for software quality, extending ISO 9126 [45]. It has been introduced in the previous learning unit: see Figure 2.3. It adds new sub-characteristics, for example:

```
Maintainability =
  ISO9126.Maintainability + {Manageability, Reusability}
```

It also adds attributes to ISO 9126: e.g. Reusability has the attributes Ratio-reused-parts and Ratio-reusable-parts, where a part is a module, a component or a sub-system. It also provides metrics to measure quality; Ratio-reused-parts, for example, is measured as the size of the reused parts divided by the size of the entire software product.

When using Quint2 as a software quality model, keep in mind that not all 32 quality attributes are equally important. A requirements engineer should prioritize the requirements.

Quality requirements should be measurable. For example, the requirement 'The system should perform well' is not measurable, but the requirement 'The response time in interactive use is less than 200 ms' is measurable.

Change scenarios

Some quality requirements do not concern functionality but other aspects of the system. These quality requirements are mainly attributes from the Maintainability and Portability group, such as Changeability and Adaptability. These requirements cannot be linked to use cases.

Such quality requirements should be linked to specific change scenarios. By doing that, you avoid being vague. For instance, instead of writing 'The system should be very portable', you should write 'The software can be installed on the Windows, Mac, and Unix platforms without changing the source code'. Instead of writing 'The system should be changeable', you should write 'Functionality that makes it possible for users to transfer money from savings to checking account can be added to the ATM within one month'.

2.3.3 Constraints

Although functional and quality requirements specify the goal, *constraints* limit the (architectural) solution space. Stakeholders should therefore not only specify requirements, but also constraints.

Possible constraint categories are the following:

- Technical constraints, such as platform, reuse of existing systems and components, use of standards;
- Financial constraints, such as budgets;
- Organizational constraints, such as processes, skill (or lack of skill) of employees, formal rules and policies;
- Time constraints, such as deadlines.

2.4 Summary requirements engineering

Summarizing: requirements engineering is a cyclic process involving domain understanding and requirements elicitation, evaluation and negotiation, requirements specification documentation, and quality assurance through requirements validation and verification. Use cases can be used to describe interaction between actors and the system. Quint2 can be used to describe the quality requirements linked to usage scenarios (use cases) or change scenarios.

Results of requirements engineering which are necessary for a software architect are descriptions of:

- the scope,

– stakeholders, concerns and their relations,
– functional requirements,
– quality requirements,
– prioritization of requirements,
– constraints.

2.5 Tactics

Once determined, the quality requirements provide guidance for architectural decisions. An architectural decision that influences the qualities of the product is sometimes called a *tactic* [5]. Mutually connected tactics are bundled together into *architectural patterns*: schemes for the structural organization of entire systems. We will discuss patterns in detail in the learning unit about patterns.

As an example of tactics, we show some tactics to achieve recoverability (a subcharacteristic of the Reliability factor in ISO 9126):

– *Voting* between processes running on redundant processors. This tactic detects only processor faults; not algorithmic errors.
– In a *hot restart*, only a few processes will experience state loss. In a "cold" restart, the entire system looses state, and is completely reloaded. In a hot restart model, the application saves state information about the current activity of the system. The information is given to the standby component so it is ready to take over quickly. There may be redundant standby components, that respond in parallel. The first response is used, the others are discarded.
– *Passive redundancy* means switching to a standby backup on failure. This tactic is often used in databases.
– *Rollback*: a consistent state is recorded in response to specific events, which makes it possible to go back to the recorded state in case of an error. This is often used in databases, or with software installation.

Some tactics for changeability (a subcharacteristic of Maintainability) are:

– Maintaining *semantic coherence*: high cohesion within every module, loose coupling to other modules;
– *Hiding information*. Provide public responsibilities through specified interfaces that are maintained when the program evolves.
– Using an *intermediary*. For example, a data repository uncouples producer and consumer; design patterns such as Facade, Mediator, and Proxy translate syntax. Brokers hide identity.

DISCUSSION QUESTIONS

1. *Buildability* is defined as the ease of constructing a desired system in a timely manner. Possible tactics are decomposition into modules with minimal coupling and assignment of modules to parallel development teams. This results in minimization of construction cost and time. Where in ISO9126/Quint2 would you locate this quality?

2. Where in ISO9126/Quint2 would you locate Scalability?

3. Can you think of other important qualities that are not explicitly mentioned in these models?

4. What tactics would you advocate to promote the Security sub-characteristic?

5. What tactics would you advocate to promote the Fault-tolerance sub-characteristic?
6. What tactics would you advocate to promote the Availability sub-characteristic?

Module 3

Description and evaluation

INTRODUCTION

During the phase of the design of the architecture, the focus is, for the first time in the development process, on the proposed solution rather than on the problem domain. The description of the architecture lays the foundation for design, by formulating rules and principles that the design should adhere to.

Architecture, which is the prudent partitioning of a whole into parts, with specific relations among the parts, is what allows a group of people - separated by organizational, geographical or temporal boundaries - to work cooperatively and productively together to solve a much larger problem than any of them would be capable of individually. Each part can be built fairly independent of the other parts. Architecture is what makes the set of parts work together as a successful whole. Architecture documentation is what tells developers how to achieve this.

Architecture involves making engineering choices that determine quality attributes. Individual goals with respect to quality are attended by specific tactics. An example of a tactic for a specific goal is: for high performance, decompose the system into cooperating processes and pay attention to the volume of interprocess communication and the frequency of data access. Another example is: for security, pay attention to authentication and authorization checks, to communication restrictions, and to vulnerability to external intrusions.

The design of the architecture should balance the concerns of many different stakeholders. Therefore, there will be no single 'best' solution. Stakeholders concerns may be contradictory, for instance, and a solution that answers one demand may conflict with other demands. These choices, and their purpose, are recorded in the architecture documentation. The documentation offers the means for an early validation: the customers get an early chance to answer the question whether the proposed system really is what they want, to solve their problem.

One of the purposes of the description of an architecture is to be able to evaluate the architecture: is the chosen architecture the right one? In general, this is not an easy question to answer. In this module, we show you different methods for describing and evaluating architectures.

LEARNING OBJECTIVES
After having studied this module you should be able to:

- summarize the contents of the IEEE 1471 standard,
- explain what a viewpoint is,
- give examples of viewpoints,
- explain what a view is, and its relationship with a viewpoint,
- explain what an architectural description language is,
- describe several approaches to describe architectures,
- describe several approaches to evaluate architectures,
- explain which steps should be carried out evaluating an architecture using the ATAM.

3.1 Describing architectures

Architecture documentation must serve many purposes. It will be used as a blueprint for construction, as a primary vehicle for communication among stakeholders, it will be used for educational purposes, to introduce people to the system, and it may be used as the basis for an evaluation.

In this section, we give some basic guidelines for architecture description, introduce the concept of different views and viewpoints, describe a standard and several frameworks for architectural description. In the next section we will discuss a number of architecture description languages.

3.1.1 Basic guidelines

– Write documentation from the reader's point of view: documents that are optimized for the convenience of the writer instead of for the efficiency for the reader, will not be read. Keep in mind that an architecture document is written once, but (if useful) read many times. To write from the reader's point of view, it is important to know who the intended audience is, and to know what they need or want to know from the document.
– Avoid almost-alike repetition. Repetitions are tedious to read, and the reader will wonder if the differences are intentional, and if so, what they signify.
– Also, avoid ambiguity (by which we mean that there are multiple possible interpretations, of which at least one is incorrect). This is an important issue, because ambiguity might lead to the wrong system being built.
– Explain the notations you use: boxes and arrows can be used to mean almost anything. A diagram with boxes and arrows without an explanation of what they stand for will therefore be ambiguous.
– Avoid an idiosyncratic organization, which will hinder the reader in navigating the document. Conform to standards if they are available.
– Explain every decision, by including a rationale. Include rejected alternatives, with a rationale. They may be of use later, for instance when decisions come under scrutiny because of new requirements.
– Do not allow your documentation to get out of date. The only exception is for temporary decisions during development.

3.1.2 Inspiration for architecture as a discipline

As the name implies, software architecture takes its inspiration from other engineering disciplines, like construction. The idea of a description of an architecture is that of a blueprint describing the architecture of a building, before the actual building begins.

From electronics, software architecture has taken the idea of standard components and interfaces, and their symbolic representation in circuit diagrams. However, there is the question whether industry standards for storing and representing information are possible.

Note, however, that it took Italian Renaissance 150 years to find a way to draw 3D perspective (see Figure 3.1): what can be the case is that we also need a bit of time before we will be able to represent all relevant concepts of a higher abstraction level.

3.1.3 Concerns and views

Figure 3.2 shows three different views of a car (for the attentive reader: the descriptions are not of the same car), for different audiences. In the same way, a software architecture is described using different views.

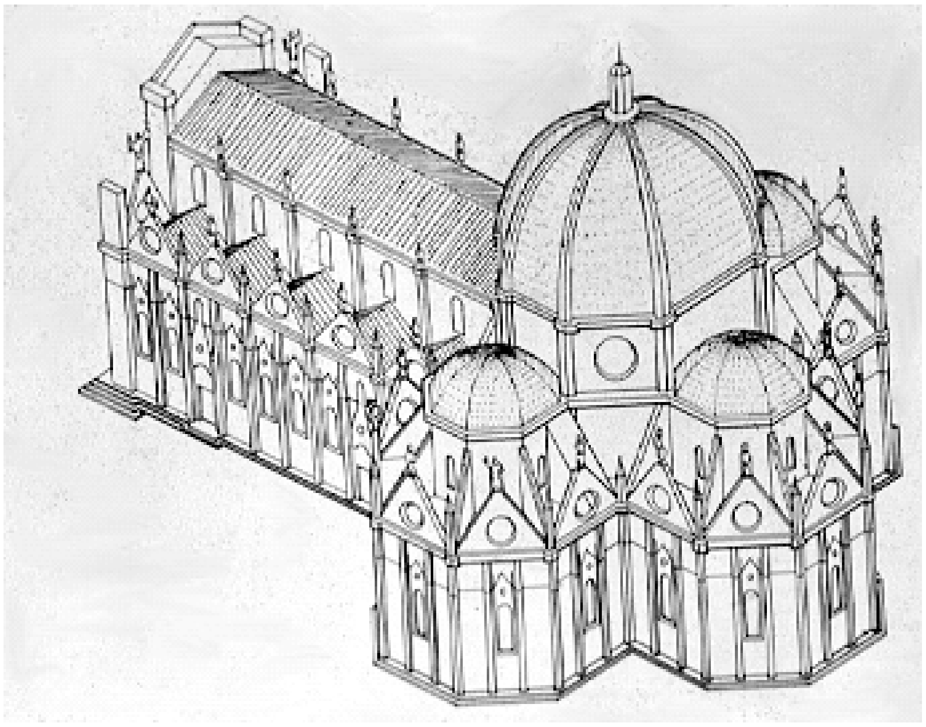

Figure 3.1: A church in 3D

A software architecture is a complex entity, that cannot be described in a simple one-dimensional fashion. There is no single rendition that is the ’right’ one. All the views together convey the architecture. It depends on your goals which views are relevant: different views appeal to different stakeholders’ concerns, and expose different quality attributes.

The essence of architecture is the suppression of information not necessary to the task at hand, and it is fitting that the very nature of architecture is that it never presents its whole self to us, but only a facet or two at a time. This is its strength: each view characterizes certain aspects of the system while ignoring or deemphasizing other aspects, all in the interest of making the problem at hand tractable.

3.1.4 Standards and frameworks for description

IEEE standard 1471

The IEEE Computer Society has developed the *IEEE Standard 1471-2000* [50], called the *Recommended Practice for Architectural Description of Software-Intensive Systems*, first published in October 2000. The IEEE 1471 is a recommended practice. An organization using the standard must decide how to employ IEEE 1471.

The standard applies to *architectural description*: architectural descriptions may conform to the standard, but systems, projects, processes or organizations cannot. The standard tells you how to describe an architecture, and is not a standard architecture, standard architectural process, or method.

The recommended practice offers a model of architectural descriptions in their context. It is written in terms of ‘shall’, ‘should’, and ‘may’, and is based on consensus. It applies

Peugeot 307, 5 doors XS 1.6-16V Automatic
Weight: 1204 kg
Max power/rpm: 110pk (80kW)/5800
Max torque/rpm.: 147Nm/3900
Top speed: 184 km/h

Figure 3.2: Three views of a car

to systems where software plays a major role. It only focuses on the organization of the descriptions; not on the systems themselves, or the actual models.

The standard does not prescribe specific languages, viewpoints, views (we will discuss the meaning of these concepts later on) or models. The standard does not contain formal consistency criteria. So, architectural descriptions may be checked for conformance to the recommended practice, but systems, projects, organizations, processes, methods or tools cannot be checked for conformance.

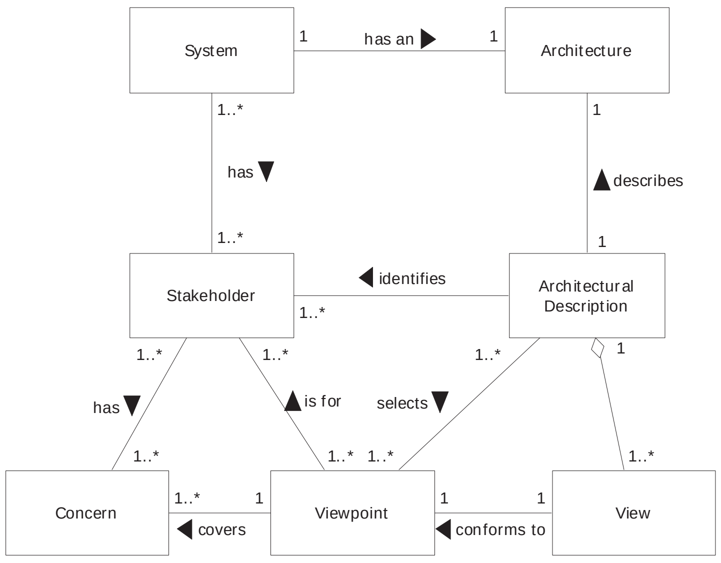

Figure 3.3: IEEE 1471 concepts

Figure 3.3 shows some concepts used in the IEEE-1471 recommended practice, and their relationships.

DEFINITION 3.1 A *stakeholder* is a person or an organization with an interest in the system. An architectural description should identify the stakeholders. A system will have at least one stakeholder, and a stakeholder may be a stakeholder of different systems. A stakeholder has at least one *concern* (an interest of a stakeholder), and each concern may be shared by different stakeholders.

DEFINITION 3.2 A *view* is a representation of the whole system from the perspective of a related set of concerns: a viewpoint. The architectural description of a system includes a number of different views. In general, several diagrams and descriptions are used for one view.

DEFINITION 3.3 A view conforms to a *viewpoint*. A viewpoint is a collection of patterns, templates, and conventions for constructing one type of view. It defines the stakeholders whose concerns are reflected in the viewpoint and the guidelines, principles, and template models for constructing its views. Viewpoints are defined with specific stakeholders' concerns in mind.

This means that a viewpoint is not related to one specific system; a viewpoint should rather be seen as a certain aspect of systems. Both views and viewpoints will be discussed in more detail.

Other concepts used in the recommended practice are model and rationale.

DEFINITION 3.4 A *model* is a piece of (structured) information in a particular representation or language.

DEFINITION 3.5 A *rationale* is an explanation or motivation for a specific choice or decision, or for a set of decisions.

To adhere to IEEE 1471-2000, an architectural description should contain the following elements:

1. For the documentation as a whole, there should be *meta-information* such as a date of issue and status, the issuing organization and the authors, a change history, a summary, a glossary and references. The context and the scope of the system should be explained.

2. *Stakeholders and their concerns* should be identified. At a minimum, the stakeholders identified include users, owners, developers and maintainers. At a minimum, the concerns identified should include the purpose or mission of the system, the appropriateness of the system for use in fulfilling its mission, the feasibility of constructing the system, the risks of system development and operation to users, owners and developers, and maintainability, deployability and evolvability of the system.

3. A selection of *architectural viewpoints* should be given. Each viewpoint is specified by a name, the stakeholders to be addressed by the viewpoint and the concerns to be addressed by the viewpoint. An important aspect is: what concepts do stakeholders recognize?

4. For each viewpoint, an architectural description should contain one or more *architectural views*. Each view corresponds to one viewpoint. A view is a representation of the system, based on a viewpoint. A view may consist of one or more models. Each view will include introductory information. In addition, formal or informal consistency and completeness tests may be given, to be applied on the models making up an associated view, evaluation or analysis techniques,

heuristics, patterns, guidelines and tips. Also, a rationale should be given, that addresses the extent to which the stakeholders and concerns are covered.

5. An architectural description should contain a *consistency analysis*. There should be a consistency analysis across views, and any known inconsistencies among views should be mentioned in the architectural description.

6. A *rationale* should be given. The rationale is the motivation for the resulting architecture and its description. Alternatives should be described, and decisions should be explained with respect to the requirements.

Several authors have described collections of viewpoints. Rozanski and Woods [43] describe the following viewpoints:

– The *functional viewpoint* describes the system's runtime functional elements and their responsibilities, interfaces, and primary interactions. The functional view of a system defines the architectural elements that deliver the system's functionality. This viewpoint addresses all stakeholders.
– The *information viewpoint* describes the way that the architecture stores, manipulates, manages, and distributes information. This viewpoint concerns both the information structure and the information flow. Stakeholders are the users, the owners and the developers.
– The *concurrency viewpoint* describes the concurrency structure of the system, mapping functional elements to concurrency units to clearly identify the parts of the system that can execute concurrently, and shows how this is coordinated and controlled. This viewpoint is concerned with the system's concurrency and state-related structure and constraints. Stakeholders are developers, testers, and some administrators.
– The *development viewpoint* describes the architecture that supports the software development process. Aspects are module organization, standardization of design and testing, instrumentation, code structure, dependencies, configuration management. Stakeholders are developers and testers.
– The *deployment viewpoint* describes the environment into which the system will be deployed, including the dependencies the system has on its runtime environment. Aspects in this viewpoint are specifications of required hardware, of required software, or network requirements. Stakeholders are system administrators, developers, testers, assessors.
– The *operational viewpoint* describes how the system will be operated, administered, and supported when it is running in its production environment. Aspects in this viewpoint are installation and upgrade, functional migration, data migration, monitoring and control, backup, configuration management and so on. Stakeholders are system administrators, developers, testers, assessors.

Other common viewpoints are:

– *Maintenance viewpoint*. Stakeholders are the maintenance engineers. Important aspects are the questions where new functionality can be added, which interfaces should be implemented, how components are to be compiled and linked etc.
– *Operations viewpoint*. Stakeholders are operators. Questions to be answered are how to install and configure the system on actual computers, how to monitor execution state, or how to shut down in case of troubles.
– *Management viewpoint*. Stakeholders are the decision makers. Questions to be answered are what kind of functionality the system will provide, how much it will cost to introduce the system in the organization, what risks are involved etc.

	What Data	How Function	Where Locations	Who People	When Time	Why Motivation
Scope (contextual) *Planner*	Things important to the business	Processes performed by the business	Locations of the business	Organizations important to the business	Events or cycles	Goals and strategies
Enterprise model (conceptual) *Owner*	Semantic model	Business process model	Business logistics system	Workflow model	Master schedule	Business plan
System model (logical) *Designer*	Logical data model	Application model	Distributed system model	Human interface model	Process structure	Business rule model
Technology model (physical) *Builder*	Physical data model	System design	Technology model	Presentation model	Control structure	Rule design
Components *Subcontractor*	Data definition	Program	Network architecture	Security architecture	Timing definition	Rule definition
Working system	Data	Function	Network	Organization	Schedule	Strategy

Table 3.1: The Zachman framework

Zachman Framework

The mission of the Zachman Institute for Framework Advancement (ZIFA) is to exercise the *Zachman Framework* for Enterprise Architecture, for the purpose of advancing the conceptual and implementation understanding of enterprise architecture [55].

The goals of the institute are to establish the Zachman Framework as a universal language to facilitate communication, research and implementation of enterprise architecture concepts. Table 3.1 shows the framework. Each row represents a point of view, each column a certain aspect, and each cell a view. In this table, the cells show possible models or documents for a given view.

The Kruchten 4+1 model

Philippe Kruchten introduced his *4+1 model* in [27]. It consists of a practical set of views for software architecture.

The main stakeholders for the *logical view* are the end users, who are concerned with functionality. Programmers are the stakeholders who are addressed in the *development view*. The *process view* is for system integrators, who are concerned with quality aspects such as performance, scalability and throughput. The *physical view* is for system engineers, who care about system topology, delivery, installation and communication issues.

The *scenarios* are used for analysis and to demonstrate completeness and consistency across the four views. Scenarios are also used to find relevant architectural concepts: they are the starting point for the architecture.

This 4+1 model is subsumed into the *Rational Unified Process* (RUP), where views correspond to UML diagram types. Concepts in the logical view are classes and services, in the development view modules and packages, in the process view components and processes, and in the physical view nodes and networks.

There are some guidelines to go from one view to another, for instance by mapping a class to a process. The 4+1 model provides useful, somewhat technical views, with an accent towards development. The model may also be used to describe existing systems.

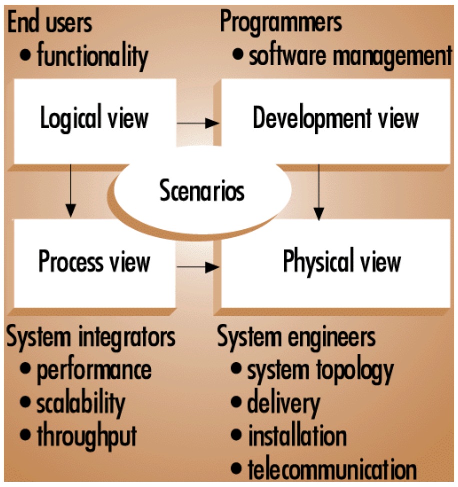

Figure 3.4: The Kruchten 4+1 model

Viewtypes, styles and views

A slightly different approach is documentation using viewtypes and styles [12]. This approach aids in deciding which diagrams to use for a certain view.

DEFINITION 3.6 A *viewtype* is a category of views, containing similar elements and relationships. A viewtype defines the element types and relationship types, used to describe the architecture of a software system from a particular perspective.

For software systems, there are three viewtypes: the module viewtype, the connector viewtype, and the allocation viewtype.

Module viewtype

The *module viewtype* is for modeling how the software system is structured as a set of implementation units. Elements in this viewtypes are modules (which may be classes or packages for instance). An element has a name, responsibilities, visible interfaces, and implementation information (such as the set of code units that implement the module). Relations within this viewtype have the form of *part-of*, *depends-on*, or *is-a*.

Several *styles* can be discerned within this viewtype: the *decomposition style* represents the decomposition of the code into systems, subsystems, subsubsystems and so on. The *uses-style* tells developers which other modules must exist for their portion of the system to correctly execute. The *generalization style* shows the inheritance relations. The *layered style* organizes the code into disjoint layers, where code in higher layers is allowed to use code in lower layers according to predefined rules.

Examples of views within this viewtype are views belonging to Rozanski and Wood's development viewpoint. Notations for this viewtype are for instance UML class diagrams and package diagrams. A criticism of class diagrams however, is that all of these relations are shown at the same time, which undercuts the usefulness of a view.

Connector viewtype

The *connector viewtype* is for modeling elements that have some run-time presence. Elements in this viewtype are runtime components: processes, objects, clients, servers, data stores, filters and so on. Another kind of elements in this viewtype are connectors: communication links, protocols, information flows, procedure calls, asynchronous messages, pipes and so on. Relations in this viewtype attach connectors to components.

Within this viewpoint, various styles may be used. Some examples are the *pipe-and-filter style*, the *shared-data-style*, the *publish-subscribe style*, the *client-server style*, the *peer-to-peer style*, or the *communicating processes style*. The correspondence of these styles to the architectural patterns that we discuss in the corresponding learning unit, shows that architectural patterns often prescribe runtime components and their relations.

An example of views within this viewtype are views belonging to Rozanski and Wood's concurrency viewpoint. Notations are for instance the UML 2.0 component diagram, the UML-RT diagram, or a data flow diagram. Note, however, that what is meant with runtime components is not exactly the same as what is meant with the components that you will read about in other learning units. Components are defined as concrete chunks of implementation (like executables or dll's), that realize abstract interfaces. UML 2.0 component diagrams are used because they use connectors and ports. Architecture description languages (we will discuss them later on) often address the connector viewtype.

Allocation viewtype

The *allocation viewtype* is for documenting the interaction of hardware, file systems and team structure with software architecture.

Three styles can be discerned within this viewtype. In the *deployment style*, elements are the processes from the connector viewtype, processors, memory, disk, and network. Relations in this style are allocated-to and migrates-to. In the *implementation style*, elements are modules, files and directories. Relations are allocated-to and containment. In the *work assignment style*, elements are modules, companies, teams and persons. Relations are allocated-to.

This viewtype van be used for views belonging to Rozanski and Wood's deployment, operational and development viewpoints. Notations for this viewtype are UML deployment diagrams, or UML artifact diagrams.

3.2 Architectural description languages

DEFINITION 3.7 An *architecture description language* (ADL) is a language (graphical or textual or both) for describing software systems in terms of its architectural elements and the relationships between them.

3.2.1 ACME

An example of an ADL is *ACME*. ACME is a language, suitable for the connector viewtype. Basic elements are components (with ports and properties) and connectors (with roles and properties). As is seen in Figure 3.5, components may be connected to a connector by attaching a port of a component to the role of a connector. The text is the equivalent of a box-and-arrow diagram, with named ports on components, and named roles on connectors. For both components and connectors, properties may be specified. However, the properties are not interpreted.

ACME was originally intended as an interchange language for ADLs. Somehow, the interchange strategy was never widely applied. Other drawbacks of ACME are: it is highly verbose, and there are no semantics.

```
System simple_client_server = {
  Component client = {
    Port send−request;
  }
  Component server = {
    Port receive−request;
  }
  Connector rpc = {
    Role caller;
    Role callee;
    Properties {
        asynchronous : boolean = true;
        max−roles : integer = 2;
    }
  }
  Attachment client.send−request to rpc.caller;
  Attachment server.receive−request to rpc.callee;
}
```

Figure 3.5: Description of a client-server system in ACME

3.2.2 Koala

Another example of an architecture description language is *Koala*. Koala is a graphical language, aimed towards describing product families of embedded software in consumer electronics (such as televisions or dvd players and recorders). This ADL is developed by Philips Electronics. In Figure 3.6, the CTVPlatform component provides

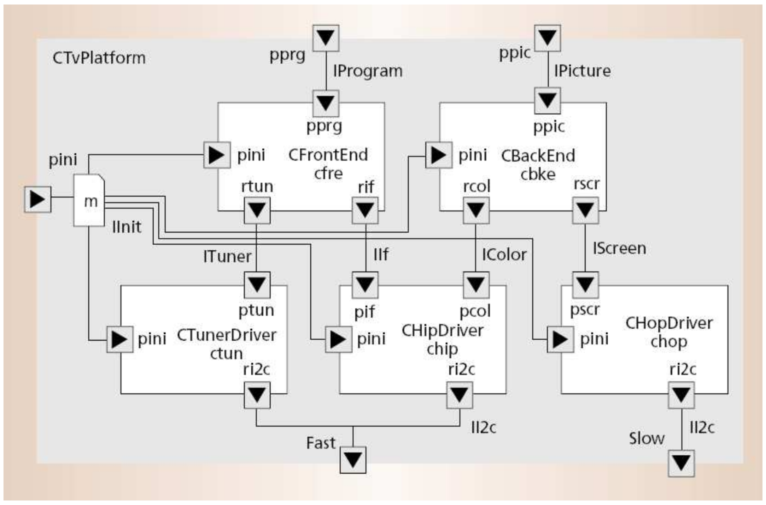

Figure 3.6: A Koala diagram

three interfaces, the pprg interface of type Iprogram, the ppic interface of type Ipicture, and the pini interface. It requires two interfaces as well, the interfaces fast and slow, of type II2c. Interface types may be described using an interface definition language:

```
interface ITuner {
        void SetFrequency(int f);
        int GetFrequency(void);
}
```

The block m in Figure 3.6 is a module that combines several interfaces. Koala compiles to C, which makes it possible to check the correspondence between provided and required interfaces.

Koala supports product lines through explicit points of variation. It allows common elements of architectures to be reused from product to product, and is still actively used at Philips.

3.2.3 Wright

```
System Foo
Component Split =
  port In = read?x -> In [] read-eof -> close -> √
  port L, R = write!x -> Out [] close -> √
  comp spec =
    let Close = In.close -> L.close -> R.close -> √
    in Close []
       In.read?x -> L.write!x ->
        (Close [] In.read?x -> R.write!x -> computation)
Component Print ...
Connector Pipe ...
Instances
  split: Split; print: Print;
  p1: Pipe
...
Attachments
  split.Left as p1.Writer;
  print.In as p1.Reader;
...
end Foo.
```

Figure 3.7: Description in Wright

In *Wright*, a system is a configuration of components interconnected by connectors, just as in ACME. In other words, Wright is an ADL to be used for the connector viewtype. Components have ports, and ports have a protocol that is specified (see Figure 3.7). The computation of the component may be specified as well. Connectors have roles, also with an associated protocol. Connectors may have an invariant as well.

Wright uses *Communicating Sequential Processes* (CSP) [23] for protocol definition. It allows for automatic checking if connected interfaces are compatible, and if the system will deadlock.

Wright has a high learning curve, because of the protocol specification. It has powerful analysis capabilities, but they are limited to a small set of properties. Wright is not actively used or supported nowadays.

3.2.4 CommUnity

CommUnity is another ADL for the connector viewtype (just as ACME and Wright). Like Wright, CommUnity offers means for describing components in terms of communication channels and actions.

```
design account is
  in add: nat
  out bal: int
  do dep: true -> bal:=bal+add
  [] wit: true -> bal:=bal-add

design regulator is
  in a: nat, b: int
  do reg: a<b ->

design channel is
  in a: nat, b: int

--------------------------------

design safeaccount is
  in add: nat
  out bal: int
  do dep: true -> bal:=bal+add
  [] wit: add<bal -> bal:=bal-add
```

Figure 3.8: Description in CommUnity

Components may be underspecified, and there is a compositional notion of refinement, as is seen in Figure 3.8, where `safeaccount` is a refinement of `account`: `safeaccount` is derived using `regulator` by identifying `a` to `add`, `b` to `ball`, and `reg` to `wit`. The semantics allows one to prove the equivalence between safeaccount and the combination of account and regulator. It is possible to augment existing designs with more channels and actions, while preserving the original properties.

Connectors are components, coordinating the interaction between other components. This can be superposed at runtime over given components.

CommUnity has rigorous mathematical semantics, which facilitates proof of many (emergent) properties, and semantic equivalence of designs. The mathematical semantics are powerful, but hard to learn and use. CommUnity allows to build complicated designs from simple components by stepwise refinement and superposition. It is the result of a new research project, and at this time, no tools are available.

3.2.5 xADL

xADL is an extensible XML-based ADL. Its common features can be reused in every domain, and new features can be created and added to the language as first-class entities. The syntax of xADL is defined in a set of XML schema's. The syntax can be extended through inheritance. xADL has an associated visualization which can be edited by the ArchStudio development environment.

xADL is still a research project, but there are already lots of tools. It can be viewed as a factory for domain-specific ADLs.

3.2.6 ADL evaluation

When evaluating ADLs for a given situation, you should ask the following questions about desired properties:

– Are the modeling constructs suitable? Some ADLs for instance offer constructs for the connector viewtype.
– Do the elements have defined semantics?
– Is it possible to analyze completeness?
– What about maintainability? Is it possible to reuse and extend models?
– Are there tools available?
– What properties can you analyze? For instance, is it possible to analyze performance?
– What about scalability? Does the language offer the possibility of composition and decomposition?
– What about understandability? Is there a high learning curve? Can you use the ADL to discuss the architecture with the stakeholders?
– Is it possible to automate conformance control?

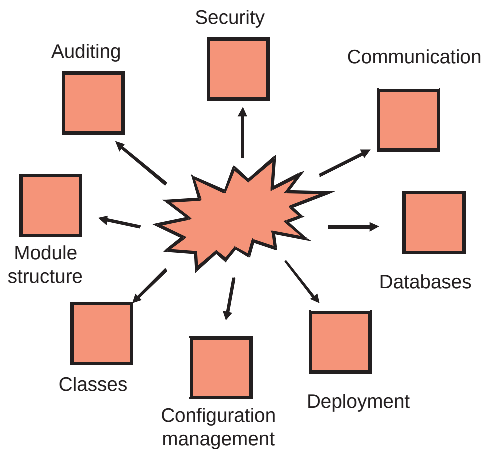

Figure 3.9: Consistency between models

Another question is whether it is possible to check the consistency between different models. Figure 3.9 shows different models, which may belong to different viewpoints and views. The question is, if there is an underlying common representation among viewpoints and views, and if there is a tool (imaginable) that can handle view derivation and guarantee consistency.

Observations about ADLs in general, are that they are occasionally used in dedicated domains. Analyzability properties can be interesting. ADLs have not (yet) hit the mainstream.

3.3 Evaluating architectures

Various terms are in use for the evaluation of architectures: assessment, analysis, review, validation. It is an active research area. The main reason why we need to evaluate architectures, is that this area is still something of a craft, without clear rules. We have to learn by doing.

Evaluating architectures is cost saving, through early detection of errors and problems. There is a reduced risk of disaster projects or acquisitions. Evaluation also increases the understanding and documentation of systems. Requirements will be clarified and prioritized. Evaluation can be seen as a form of organizational learning. But there are also costs: reviewer time, development team time and focus, and time of other stakeholders. For an analysis of costs and benefits, see [1].

3.3.1 Basic issues

When? There are several approaches to the question when to evaluate:

- regularly, as part of an iterative architecting process,
- early, to validate the overall course of the project,
- *toll gate*: a check before major construction starts, or before a supplier is selected,
- (too) late, to determine how to resolve architectural problems.

Issues here are whether evaluation is done formally or informally, and whether evaluation takes place planned versus unplanned.

Who? For the question who performs the evaluation, there are also several possible answers:

- the development team itself,
- another group in the same organization,
- external reviewers, preferably with no interest in the outcome of the evaluation.

Issues here are the questions of domain knowledge and experience: the development team will have both, the other group in the organization will probably only have domain knowledge, while a group of experts probably only will have experience. Other issues are competition and bias.

What? Several aspects of an architecture can be evaluated: the architecture process, the description, overall architectural issues such as completeness, specific quality areas, or future changes and the impact they will have on the architecture.

Issues here are: it is important to set a clear goal, and to get the right review team for what you want to evaluate.

How? The techniques for evaluating architectures can be divided in two sets:

- Questioning techniques. Possible methods are to generate discussion, based on qualitative questions, to use questionnaires, to use checklists, or to use scenarios.
- Measuring techniques. Possible methods are to provide quantitative answers to specific questions, to use metrics, to use simulation, or make use of prototypes, or to use formal analysis.

3.3.2 Review of architecture description and process

The focus of this approach is the question if the architecture description meets the standards. In this way, one asks implicitly if the architect(s) did perform the right activities.

The method for this approach is:

- Determine the goal.
- Gather the relevant documents.
- Read and comment.
- Analyze what you read using the description framework of IEEE 1471. For certain models or documents, you may use more specific checklists.
- Optionally, discuss your comments with relevant parties.
- Report your observations, risks and suggested improvements.

Typical issues to check are:

- Have all stakeholders been considered?

- Have the main requirements been identified?
- Have the quality issues been considered and selected?
- Which concerns have been identified?
- Which viewpoints have been defined?
- Is there a rationale for the viewpoints being defined?
- How are the views documented? Is there conformance to certain modeling techniques or languages?
- What is the quality of the models and the documents, in terms of readability, organization, terminology?
- Is there a rationale for the proposed solution?
- How are the documents managed?
- Can you say something about consistency between models?

3.3.3 Scenario-based analysis

DEFINITION 3.8 *Scenarios* are brief narratives of expected or anticipated use of a system from both development and end-user viewpoints.

DEFINITION 3.9 A *sensitivity point* is an architectural decision involving one or more architectural components and/or connections, that is critical for achieving a particular quality attribute response measure [13].

DEFINITION 3.10 A *trade-off point* is an architectural decision that affects more than one attribute and is a sensitivity point for more than one attribute ([13]). For example, changing the level of encryption could have a significant impact on both security and performance, in an opposite direction.

Scenario-based analysis can be used for different goals:
- Identify major risks. In that case, you use likely scenarios with major impact, or you search for scenarios that are not handled in the architecture. By using those scenarios to analyze the architecture, you identify sensitivity and trade-off points.
- Validate quality goals. The typical quality goals that you validate using scenario-based analysis are maintainability and modifiability. The scenarios you use are change scenarios, that are likely to be needed in a particular period. During the analysis, you estimate the cost of realizing changes.
- Choose between architectural alternatives. Scenarios can be used to compare scores among multiple candidates. Architectures are being scored with respect to minimal risk, and the estimated effort for maintenance.

3.3.4 Architecture Tradeoff Analysis Method (ATAM)

DEFINITION 3.11 ATAM is an abbreviation for *Architecture Tradeoff Analysis Method*. It is a scenario-based, standardized evaluation method.

ATAM focuses on trade off's between quality goals. This method requires the participation of three different groups: the evaluation team (a group of three to five people, external to the project), the project decision makers (including the architect), and the architecture stakeholders (developers, testers, users, ...).

Deliverables of this method are:
- quality requirements in terms of scenarios,
- a mapping of architectural decisions to quality requirements,

- a set of identified sensitivity and trade off points,
- a set of identified risks and overarching risk themes.

Steps to take within ATAM are:

1. Present the ATAM (dome by the evaluation leader).
2. Present the business drivers (done by the decision makers).
3. Present the architecture (done by the lead architect).
4. Identify the architectural approaches and patterns used (done by the architect).
5. Generate quality attribute utility tree (the quality attributes that comprise system 'utility', such as performance, availability, security, modifiability, usability and so on, are elicited, specified down to the level of scenarios, annotated with stimuli and responses, and prioritized).
6. Analyze the architectural approaches (based upon the high-priority scenarios, the architectural approaches that address those scenarios are elicited and analyzed).
7. Brainstorm and prioritize scenarios (all stakeholders) (a larger set of scenarios is elicited from the entire group of stakeholders. This set is prioritized and analyzed).
8. Analyze the architectural approaches for a second time.
9. Present the results.

DISCUSSION QUESTIONS

1. In RUP, Kruchten's '4+1'- model is combined with UML as a description language. Argue whether this is the only choice, and, if not, whether this is the best choice.

2. Perspectives [44] are another concept for the description of an architecture. Discuss the relationship between perspectives and viewpoints. Would you typically use only viewpoints, or only perspectives, or both, in an architectural description?

3. How would you make a cost-benefit argument for the inclusion or exclusion of a particular view in the documentation? If you could summon up any data to support your case, what data would you want?

4. The reason for evaluating architectures in the first place is that architecture is still something of a craft, without firm rules. But does not the same problem occur in architecture evaluation? If so, is evaluation still useful and why?

Block II

Patterns in software architecture

Module 4

Architectural patterns

INTRODUCTION

Before major development starts, we need to choose an architecture that will provide us with the desired quality attributes. Therefore we need a way to discuss architectural options and their quality consequences in advance, before they can be applied to a design. Our decisions at this stage can only be based on experience with architectural choices in previous systems. Such architectural experience, reduced to its essence and no longer cluttered with the details of the systems that produced it, is recorded in architectural patterns.

Many systems have a similar structure. Distributed systems for instance, can have a client-server structure, with clients making requests, and a server processing those requests and answering them. When we observe a similarity, we want to know what is common among the solutions, and what variations are possible. We ask in what circumstances an approach may be used, and how it should be used and customized for a particular system. An architectural pattern provides an answer to such questions.

This module introduces architectural patterns and their relation with design patterns. We will discuss several architectural patterns: the layers pattern, the client-server pattern, the master-slave pattern, the pipe-filter pattern, the broker, the peer-to-peer pattern, the event-bus, model-view-controller, the blackboard, and the interpreter.

These examples do not exhaust all architectural patterns: we will discuss ways to classify them. With an example, we will show the effect of choosing different patterns for the same problem. The example makes clear that you need to be able to choose the right pattern for a given situation. To make the right choice you need experience with different systems. Knowledge about patterns will help you to expand upon what you learn by experience, because by learning about patterns, you learn from the experience of others.

LEARNING OBJECTIVES
After having studied this module you should be able to:
- to describe the structure and function of the patterns which are covered in this module,
- to describe the advantages and disadvantages of the patterns which are covered in this module,
- to explain the difference between a style and a pattern,
- to give examples of applications of the patterns covered in this module,
- to name examples of patterns fitting in a certain style.

4.1 Architectural patterns or styles

DEFINITION 4.1 An *architectural pattern* is a proven structural organization schema for software systems.

A pattern is a description of a set of predefined subsystems and their responsibilities. In a system structured according to the client-server pattern, for instance, two subsystems are distinguished: the client (which can have many instances), and the server (which is unique). The responsibility of the client may be to show a user-interface to the user; the responsibility of the server may be to process lots of questions, and to guard data that are of interest to the client.

A pattern also describes rules and guidelines for organizing the relationships among the subsystems. The relationship between client and server is that the client asks questions and the server answers them.

Patterns are written by people with lots of experience. Patterns make knowledge which could have remained hidden in the heads of these experienced people explicit. This enables others to learn from those experiences. Patterns are not constructed by a single person: they reflect the experience of many developers. They capture existing, well-proven solutions in software development, and help to promote good design practices.

Architectural patterns are also called *styles*, or standard architectures, but the word architectural style is more often used for a concept less fine-grained than a pattern; several patterns may then belong to the same architectural style. We will explain the subtle differences later in this module.

4.1.1 Why are patterns helpful?

When a certain kind of problem is solved by many developers in a similar way, and it is generally accepted that this way solves that problem well, it becomes a pattern. So, a pattern addresses a recurring design problem, for which a general solution is known among experienced practitioners: a pattern documents existing, well-proved design solutions.

By writing a pattern, it becomes easier to reuse the solution. Patterns provide a common vocabulary and understanding of design solutions. Pattern names become part of a widespread design language. They remove the need to explain a solution to a particular problem with a lengthy description. Patterns are therefore a means for documenting software architectures. They help maintaining the original vision when the architecture is extended and modified, or when the code is modified (but can not guarantee that).

Patterns support the construction of software with defined properties. When we design a client-server application, for instance, the server should not be built in such a way that it initiates communication with its clients.

Many patterns explicitly address non-functional requirements for software systems. For example, the MVC (Model-View-Controller) pattern supports changeability of user interfaces. Patterns may thus be seen as building blocks for a more complicated design.

4.1.2 Pattern schema or template

Patterns are described using a *pattern template* or *schema*. All of the many different templates have at least the following components:

- *Context*: the situation giving rise to a problem.
- *Problem*: the recurring problem in that context. A solution to the problem should fulfill requirements, consider constraints, and have desirable properties. These conditions are called forces. Forces may conflict with each other (performance

may conflict with extensibility, for instance). Forces differ in the degree in which they are negotiable.
- *Solution*: a proven solution for the problem. The solution is given as a structure with components and relationships, and as a description of the run-time behavior. The first description is a static model of the solution; the second is a dynamic one.

4.1.3 Design patterns and architectural patterns

What is the difference between *design patterns* and architectural patterns? Design patterns offer a common solution for a common problem in the form of classes working together. They are thus smaller in scale than architectural patterns, where the components are subsystems rather than classes.

Design patterns do not influence the fundamental structure of a software system. They only affect a single subsystem. Design patterns may help to implement an architectural pattern. For example, the observer pattern (a design pattern) is helpful when implementing a system according to the MVC architectural pattern.

The concept of patterns was originally introduced by Christopher Alexander in building architecture, in 'A pattern language' [3], and 'The timeless way of building' [2]. Design patterns first attracted attention in software design and development (the *Gang of Four* book [20]). Since then, patterns have been used in more disciplines: there are analysis patterns, user interface patterns, programming idioms, functional design patterns, and so on.

4.2 Examples of architectural patterns

In this section, we describe several architectural patterns. For each we describe the components and connections involved, give one or more usage examples, and discuss advantages, disadvantages and other issues.

4.2.1 Layers pattern

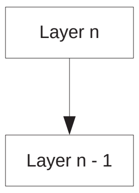

Figure 4.1: Layers of different abstraction levels

The *layers architectural pattern* (see Figure 4.1) helps to structure applications that can be decomposed into groups of subtasks, each of which is at a particular level of abstraction. Each layer provides services to the next higher layer. Services in a layer are implemented using services from the next lower layer. Service requests are frequently done by using synchronous procedure calls.

In short, this pattern means that conceptually different issues are implemented separately, and that layers of a higher abstraction level use services of a lower abstraction level, and not the other way around.

This pattern has the following benefits:

- A lower layer can be used by different higher layers. The TCP layer from TCP/IP connections for instance, can be reused without changes by various applications such as telnet or FTP.
- Layers make standardization easier: clearly defined and commonly accepted levels of abstraction enable the development of standardized tasks and interfaces.
- Dependencies are kept local. When a layer shows the agreed interface to the layer above, and expects the agreed interface of the layer below, changes can be made within the layer without affecting other layers. This means a developer can test particular layers independently of other layers, and can develop them independently as well: development by teams is supported this way.

A result of the above is that layers may easily be replaced by a different implementation.

Example: networking protocols

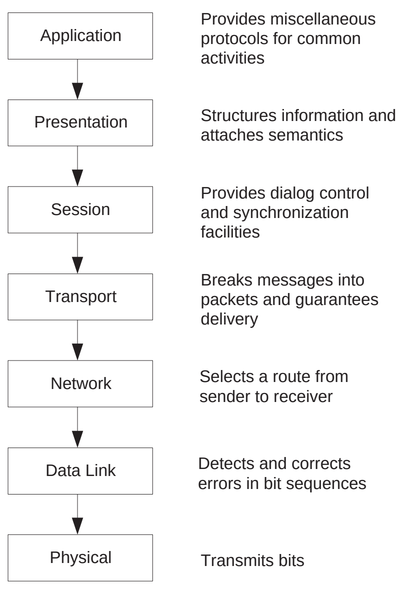
Application
Provides miscellaneous protocols for common activities
Presentation
Structures information and attaches semantics
Session
Provides dialog control and synchronization facilities
Transport
Breaks messages into packets and guarantees delivery
Network
Selects a route from sender to receiver
Data Link
Detects and corrects errors in bit sequences
Physical
Transmits bits

Figure 4.2: Example: networking protocols

Figure 4.2 depicts the ISO Open System Interconnect seven layer protocol (the *ISO/OSI protocol*). TCP/IP is a simplified version of this protocol. FTP and HTTP are examples from the application layer; TCP is the transport layer, and IP is the network layer.

Some other examples of this pattern are:

- In the Java virtual machine the application in Java consists of instructions for the Java virtual machine; the JVM uses services from the operating system underneath.
- The standard C library is built on Unix system calls.
- Web application systems often show four or more layers: presentation, application logic, domain logic, and data.
- The microkernel architecture has layers on top of the microkernel: The Mach operating system, the JBoss application server and Apache Derby are examples of the microkernel architecture.

Issues in the Layers pattern

The most stable abstractions are in the lower layer: a change of the behavior of a layer has no effect on the layers below it. The opposite is true as well: a change of the behavior of a lower layer has an effect on the layers above it, so should be avoided.
Of course, changes of or additions to a layer without an effect on behavior will not affect the layers above it. Layer services can therefore be implemented in different ways (think of the bridge pattern here, where a dynamic link is maintained between abstraction and implementation).
Layers can be developed independently. However, defining an abstract service interface is not an easy job. There may also be performance overhead due to repeated transformations of data. Furthermore, the lower layers may perform unnecessary work, not required by the higher layer.
There are several variants of the Layers pattern.

- In a *Relaxed layered system*, each layer may use the services of all layers below it, not only of the next lower layer. This has efficiency benefits, but leads to a loss of maintainability. An example is formed by the user interface of Eclipse: Eclipse makes use of the SWT (the Standard Widget Toolkit, [15]) and JFace [14]. JFace is a layer on top of the SWT and offers a higher level of abstraction than SWT, but it is possible to use classes of both JFace and the SWT in the same application.
- Another variant is to allow *callbacks* for bottom-up communication: here, the upper layer registers a callback function with the lower layer, to be notified at the occurrence of an events.

4.2.2 Client-server pattern

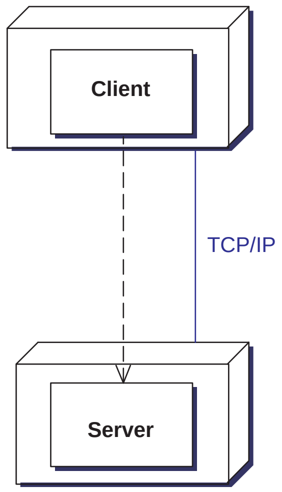

Figure 4.3: Client-server pattern

In the Client-server architectural pattern (see Figure 4.3), a server component provides services to multiple client components. A client component requests services from the server component. Servers are permanently active, listening for clients.
The requests are sent beyond process and machine boundaries. This means that some inter-process communication mechanism must be used: clients and servers may reside on different machines, and thus in different processes. In fact, you can see the Client-server pattern as a form of the layered pattern, crossing process or machine boundaries: clients form the higher level and the server forms the lower level.

Examples

Examples of the Client-server pattern are remote database access (client applications request services from a database server), remote file systems (client systems access files, provided by the server system; applications access local and remote files in a transparent manner) or web-based applications (browsers request data from a web server).

Issues in the Client-server pattern

Requests are typically handled in separate threads on the server.

Interprocess communication causes overhead. Requests and result data often have to be transformed or marshaled because they have a different representation in client and server, and there is network traffic.

Distributed systems with many servers with the same function should be transparent for clients: there should be no need for clients to differentiate between servers. When you type in the URL for Google for instance, you should not have to know the exact machine that is accessed (location transparency), the platform of the machine (platform transparency), or the route your request travels, and so on. Intermediate layers may be inserted for specific purposes: caching, security, load balancing for instance.

Sometimes, callbacks are needed for event notification. This can also be seen as a transition to the Peer-to-peer pattern.

State in the Client-server pattern

Clients and servers are often involved in 'sessions'. This can be done in two different ways:

- With a *stateless server*, the session state is managed by the client. This client state is sent with each request. In a web application, the session state may be stored as URL parameters, in hidden form fields, or by using cookies. This is mandatory for the REST architectural style [18] for web applications.
- With a *stateful server*, the session state is maintained by the server, and is associated with a client-id.

State in the Client-server pattern influences transactions, fault handling and scalability. Transactions should be atomic, leave a consistent state, be isolated (not affected by other requests) and durable. These properties are hard to obtain in a distributed world.

Concerning fault handling, state maintained by the client means for instance that everything will be lost when the client fails. Client-maintained state poses security issues as well, because sensitive data must be sent to the server with each request. Scalability issues may arise when you handle the server state in-memory: with many clients using the server at the same time, many states have to be stored in memory at the same time as well.

REST architecture

REST stands for Representational State Transfer. A REST architecture is a client-server architecture, where clients are separated from servers by a uniform interface. Communication is stateless. A server may be stateful, but in that case, each server-state should be addressable (for instance, by a URL). Servers offer addressable Resources. A REST architecture is also a layered system: for a client, it is transparent whether it is directly connected to a server, or through one or more intermediaries.

Web applications with stateless communication follow the rules of this pattern.

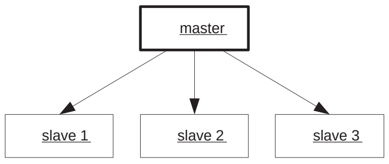

Figure 4.4: Master-slave pattern

4.2.3 Master-slave pattern

The *Master-slave pattern* (see Figure 4.4) supports fault tolerance and parallel computation. The master component distributes the work among identical slave components, and computes a final result from the results the slaves return. Figure 4.5 shows a sequence diagram of a master distributing work between slaves.

The Master-slave pattern is applied for instance in process control, in embedded systems, in large-scale parallel computations, and in fault-tolerant systems.

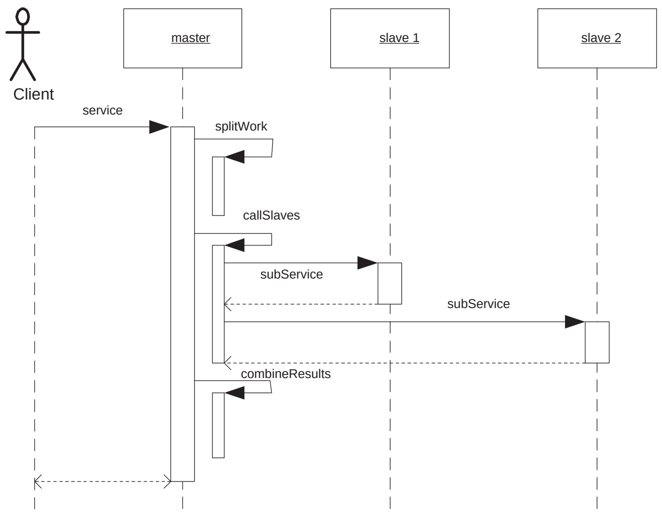

Figure 4.5: Sequence diagram for the master-slave pattern

Examples

An application area for the Master-slave pattern is fault tolerance: the master delegates the job to be done to several slaves, receives their results, and applies a strategy to decide which result to return to the client. One possible strategy is to choose the result from the first slave that terminates. Another strategy is to choose the result that the majority of slaves have computed. This is fail-proof with respect to slaves (the master can provide a valid result as long as not all slaves fail), but not with respect to the master. Failure of slaves may be detected by the master using time-outs. Failure of the master means the system as a whole fails.

Another application area is parallel computing: the master divides a complex task into a number of identical subtasks. An example is matrix computation: each row in the product matrix can be computed by a separate slave.

A third application area is that of computational accuracy. The execution of a service is delegated to different slaves, with at least three different implementations. The master waits for the results, and applies a strategy for choosing the best result (for instance the average, or the majority).

Issues in the Master-slave pattern

The Master-slave pattern is an example of the *divide-and-conquer* principle. In this pattern, the aspect of coordination is separated from the actual work: concerns are separated. The slaves are isolated: there is no shared state. They operate in parallel.

The latency in the master-slave communication can be an issue, for instance in real-time systems: master and slaves live in different processes.

The pattern can only be applied to a problem that is decomposable.

4.2.4 Pipe-filter pattern

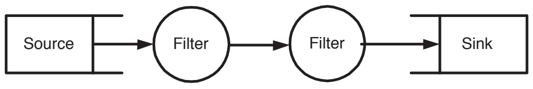

Figure 4.6: Pipe-filter pattern

The *Pipe-filter architectural pattern* (see Figure 4.6) provides a structure for systems that produce a stream of data. Each processing step is encapsulated in a filter component (a circle in Figure 4.6). Data is passed through pipes (the arrows between adjacent filters). The pipes may be used for buffering or for synchronization.

Examples

Examples of the Pipe-filter pattern are *Unix shell commands*, such as:

```
cat file | grep xyz | sort | uniq > out
```

This pattern divides the task of a system into several sequential processing steps. The steps are connected by the data flow through the system: the output of a step is the input for the next step. In the example, the `cat` filter reads the file and passes the contents of the file to the `grep` filter. The `grep` filter selects lines containing `xyz`, and passes these lines to the `sort` filter. The `sort` filter sorts the lines, and passes the sorted lines to the `uniq` filter. The `uniq` filter removes duplicate lines, and passes the result to `out`.

A filter consumes and delivers data incrementally (another name for the same concept is lazily): it produces output as soon as it comes available, and does not wait until all input is consumed.

Another example of the Pipe-filter pattern is formed by compilers. A lexical analyzer analyses the source file and sends the resulting tokens to a parser, which sends the resulting parse tree to a semantic analyzer, which produces an augmented syntax tree, that is used by the code generator to produce code, which may be optimized, and ultimately translated into machine code (there may be more or fewer steps involved). Figure 4.7 shows the components of a compiler. In practice, compilers do not follow the pattern strictly: the steps share global data (the symbol table).

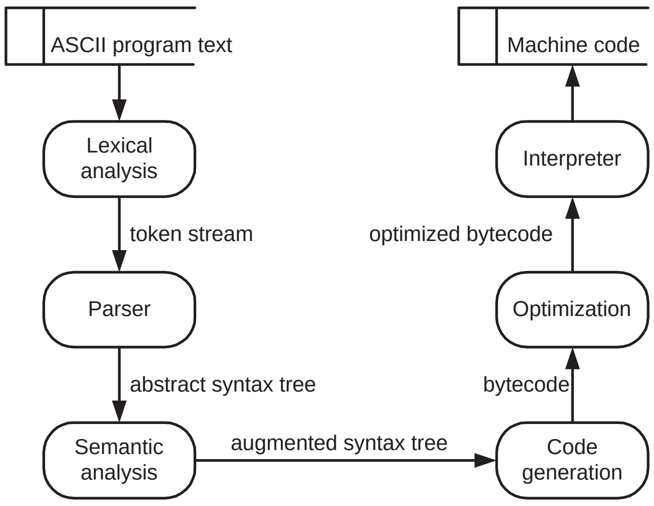

Figure 4.7: A compiler

Issues in the Pipe-filter pattern

The Pipe-filter pattern has several nice properties, and some disadvantages. It is easy to add new filters: a system built according to the pipe-filter pattern is easy to extend. Filters are reusable: it is possible to build different pipelines by recombining a given set of filters. Because of the standard interface, filters can easily be developed separately, which is also an advantage. However, that same interface may be the cause of overhead because of data transformation: when the input and the output have the form of a string for instance, and filters are used to process real numbers, there is a lot of data-transformation overhead.

Filters do not need to store intermediate results in files, and need not share state. Input and output can come from, and go to different places. Another advantage of the Pipe-filter pattern is that it shows natural concurrent processing, when input and output consist of streams, and filters start computing when they receive data. Analysis of the behavior of a pipe-filter-based system is easy, because it is a simple composition of the behaviors of the filters involved. When the input is called `x`, the behavior of the first filter is described by function `g`, and the behavior of the second filter is described by function `f`, the result of the pipeline can be described as:

```
f(g(x))
```

Because of this composition property, it is possible to analyze throughput as well (throughput is determined by the slowest filter), and the possibility of deadlocks. Deadlocks may occur when at least one of the filters needs all data before producing output. If a filter does, the size of the buffer may be too small, and the system may deadlock. The Unix `sort` filter is an example of such a filter.

A disadvantage of the Pipe-filter pattern, aside from the already mentioned possible data-transformation overhead, is that it is hard to use it for interactive applications.

4.2.5 Broker pattern

The *Broker pattern* is used to structure distributed systems with decoupled components, which interact by remote service invocations. Such systems are very inflexible when components have to know each others' location and other details (see Figure 4.8). A broker component is responsible for the coordination of communication among components: it forwards requests and transmits results and exceptions.

Servers publish their capabilities (services and characteristics) to a broker. Clients request a service from the broker, and the broker then redirects the client to a suitable service from its registry.

Using the Broker pattern means that no other component than the broker needs to focus on low-level interprocess-communication.

Interface Definition Language (IDL)

To give a textual description of the interfaces a server offers, an *Interface Definition Language* (IDL) is used. Examples of IDLs are OMG-IDL (Object Management Group, for CORBA), Open Service Interface Definitions, or WSDL (Web Service Description Language).

Alternate software may use a binary standard, like Universal Network Objects for the OpenOffice suit. Such a binary description consists of a repository which binds service names to components. It allows clients to use components indirectly, using those service names. A binary standard needs support from the programming language used.

Examples

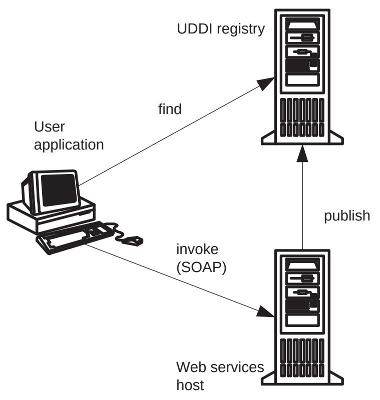

Figure 4.8: Web services

An example in which the Broker pattern is used is formed by *web services*. The Broker pattern in a web services application, as shown in Figure 4.8, is the server with the *UDDI registry*. UDDI stands for Universal Discovery, Description and Integration. It is a repository of web services. The IDL used for web services is WSDL. SOAP (Simple Object Access Protocol) is the transport protocol for messages, written in XML.

Another example of the Broker pattern is *CORBA*, for cooperation among heterogeneous object-oriented systems, and web services.

Issues in the Broker pattern

The Broker pattern allows dynamic change, addition, deletion and relocation of objects, and it makes distribution transparent to the developer. It requires standardization of service descriptions. When two brokers cooperate, bridges may be needed to hide implementation details, as in Figure 4.9:

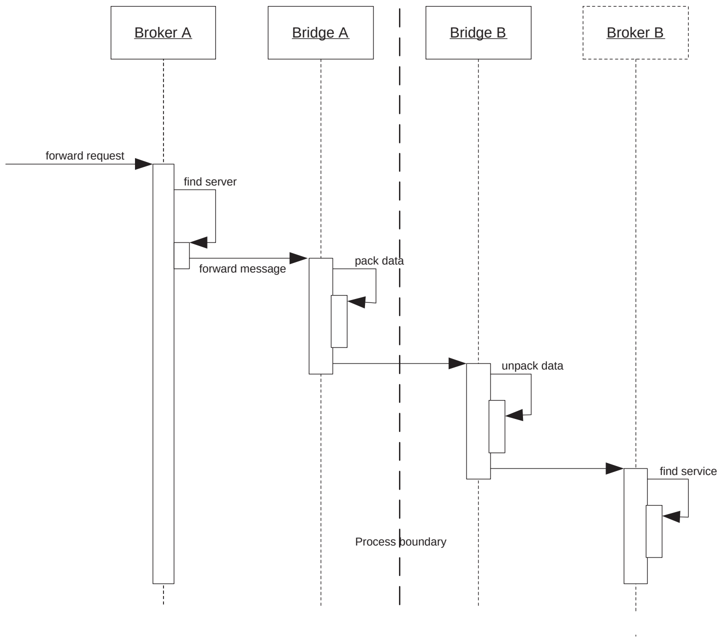

Figure 4.9: Two brokers cooperating

Broker A receives an incoming request for a certain service. It locates the server responsible for executing the specified service by looking it up in the repositories. Since the corresponding server is available at another network node, the broker forwards the request to a remote broker. The message is passed from broker A to bridge A. This component is responsible for converting the message from the protocol defined by broker A, to a network-specific but common protocol understandable by the two participating bridges. After message conversion, bridge A transmits the message to bridge B. Bridge B maps the incoming request from the network-specific format to a broker B-specific format. Broker B then performs all the actions necessary when a request arrives, as described in the first step of the scenario.

4.2.6 Peer-to-peer pattern

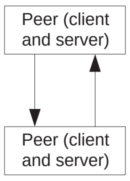

Figure 4.10: Peer-to-peer pattern

The *Peer-to-peer pattern* can be seen as a symmetric Client-server pattern: peers may function both as a client, requesting services from other peers, and as a server, providing services to other peers. A peer may act as a client or as a server or as both, and it may change its role dynamically.

Both clients and servers in the Peer-to-peer pattern are typically multithreaded. The services may be implicit (for instance through the use of a connecting stream) instead of requested by invocation.
Peers acting as a server may inform peers acting as a client of certain events. Multiple clients may have to be informed, for instance using an event-bus.

Examples

Examples of the Peer-to-peer pattern are the domain name system for internet, the distributed search engine Sciencenet, multi-user applications like a drawing board, or peer-to-peer file-sharing like Gnutella [52].

Issues in the Peer-to-peer pattern

An advantage of the peer-to-peer pattern is that nodes may use the capacity of the whole, while bringing in only their own capacity. In other words, there is a low cost of ownership, through sharing. Also, administrative overhead is low, because peer-to-peer networks are self-organizing.
The Peer-to-peer pattern is scalable, and resilient to failure of individual peers. Also, the configuration of a system may change dynamically: peers may come and go while the system is running.
A disadvantage may be that there is no guarantee about quality of service, as nodes cooperate voluntarily. For the same reason, security is difficult to guarantee. Performance grows when the number of participating nodes grows, which also means that it may be low when there are few nodes.

4.2.7 Event-bus pattern

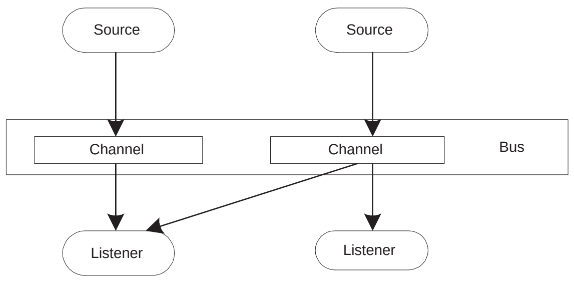

Figure 4.11: Event-bus pattern

The *Event-bus pattern* is a pattern that deals with events. It works as follows: event sources publish messages to particular channels on an event bus. Event listeners subscribe to particular *channels*. Listeners are notified of messages that are published to a channel to which they have subscribed.
Generation and notification of messages is asynchronous: an event source just generates a message and may go on doing something else; it does not wait until all event listeners have received the message.
Channels may be implicit, for instance using the event pattern, implemented in the Java event model. An explicit channel means that a subscriber subscribes directly at specific named publisher; an implicit channel means that a subscriber subscribes to a specific named channel (to a particular event type in the Java event model), and does not need to know which producers produce for that channel.

Examples

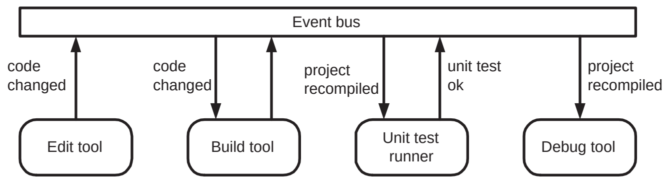

Figure 4.12: Software development environment

The Event-bus pattern is used in process monitoring, in trading systems, and in software development environments as is shown in Figure 4.12. Another example is real-time data distribution middleware, like OpenSplice [40].

Issues in the Event-bus pattern

The Event-bus pattern has the following characteristics. New publishers, subscribers and connections can be added easily, possibly dynamically. Delivery issues are important: for the developer of an event listener, it is important to realize that assumptions about ordering, distribution, and timeliness are hard to make. Also, scalability may be a problem, as all messages travel through the same event bus: with an increase of the number of messages, the capacity of the event bus may become the bottleneck.

The Event-bus pattern also allows several variations. The bus can provide event transformation services, for instance, or the bus can provide coordination, to script specific tasks.

4.2.8 Model-view-controller pattern

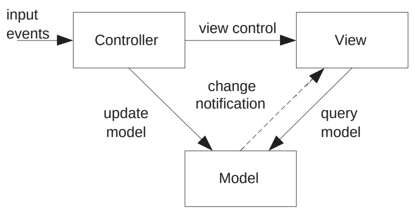

Figure 4.13: Model-view-controller

In the *Model-View-Controller pattern*, or MVC pattern (see Figure 4.13), an interactive application is divided into three parts: the model contains the core functionality and data, the view displays the information to the user (more than one view may be defined), and the controller handles the input from the user.

The MVC pattern is particularly suitable for multiple graphical user interfaces (GUIs). The model does not depend on the number and kind of GUIs, so the pattern allows for easy changes to the "look and feel".

Consistency between model and view is maintained through notification. The MVC pattern often uses the *observer design pattern*. User input can invoke a change in the model, and a subsequent change in what the view displays, as is shown in the sequence diagram of the MVC pattern in Figure 4.14.

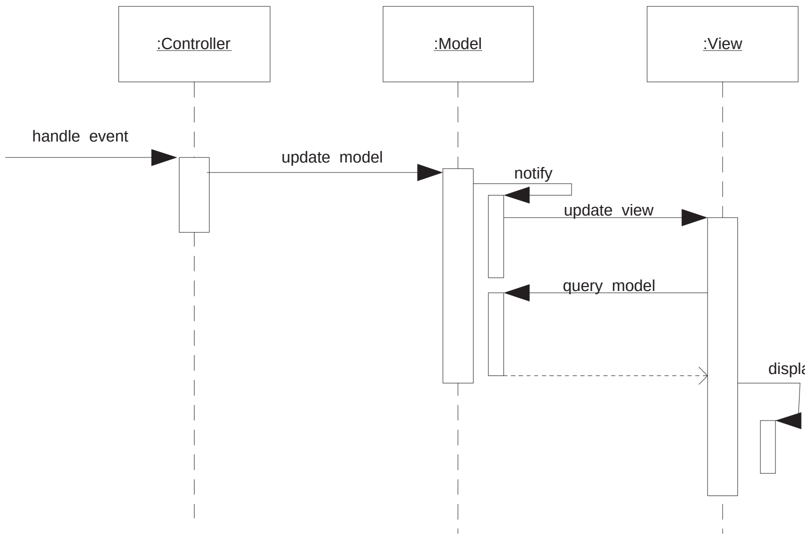

Figure 4.14: Sequence diagram of the MVC pattern

Examples

The MVC pattern was introduced with the Smalltalk programming language. It was called a paradigm then: the concept of pattern did not exist.

Examples of the MVC pattern are web presentation (see the module on enterprise application architecture), and the document-view architecture of Windows applications, which enables users to see for instance OpenOffice Writer or Impress documents in different views (think of print layout, web layout, overview).

Issues in the MVC pattern

The MVC pattern makes it easy to have multiple views of the same model, which can be connected and disconnected at run-time. It is possible to base an application framework on this pattern. Smalltalk development environments already did this.

However, the MVC pattern increases complexity. Not all visible elements lend themselves for separation of model, views and control: menus and simple text elements may be better off without the pattern. Also, the pattern potentially leads to many unnecessary updates, when one user action results in different updates.

View and control are separated, but are very closely related. In practice, they are often put together. Views and controllers are also closely coupled to the model. In web applications, a change in the model (for instance adding an e-mail address to data about persons) will lead to a change in the view and controller as well (the web site will have to show the property, and the possibility to change the property should be added as well).

4.2.9 Blackboard pattern

The *Blackboard pattern* is useful for problems for which no deterministic solution strategies are known. Several specialized subsystems assemble their knowledge to build a possibly partial or approximate solution.

All components have access to a shared data store, the blackboard. Components may produce new data objects that are added to the blackboard. Components look for particular kinds of data on the blackboard, and may find these by pattern matching.

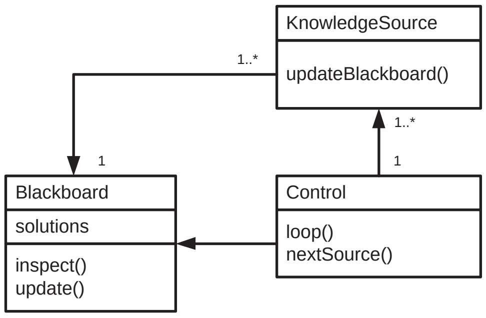

Figure 4.15: Blackboard pattern

Examples

Examples of problems in which the Blackboard pattern can be usefully applied are speech recognition, submarine detection, or inference of the 3D structure of a molecule. Tuple Space systems, like JavaSpaces, form another example of this pattern.

Issues in the blackboard pattern

Adding new applications is easy. Extending the structure of the data space is easy as well, but modifying the structure of the data space is hard, as all applications are affected. Furthermore, processes have to agree on the structure of the shared data space. There may be a need for synchronization and access control.

4.2.10 Interpreter pattern

The *Interpreter pattern* is used for designing a component that interprets programs written in a dedicated language. The interpreted program can be replaced easily.

Examples

Examples of the interpreter pattern are rule-based systems like expert systems, web scripting languages like JavaScript (client-side) or PHP (server-side), and Postscript.

Issues in the Interpreter pattern

Because an interpreted language generally is slower than a compiled one, performance may be an issue. Furthermore, the ease with which an interpreted program may be replaced may cause a lack of testing: stability and security may be at risk as well.

On the other hand, the interpreter pattern enhances flexibility, because replacing an interpreted program is indeed easy.

4.3 Architectural styles

Patterns have been developed bottom-up: for a given problem, a certain kind of solution has been used over and over again, and this solution has been written down in the form of a pattern.

Architectural styles on the other hand, have been formulated top-down: when you see a software system as a configuration of components and connectors, you can classify them according to the nature of the components and connectors. In general, patterns will belong to one of those styles.

Mary Shaw and Paul Clements have proposed a classification of styles [47], based on the constituent parts (components and connectors), control issues and data issues. This classification is as follows:

- *Interacting processes* have their own thread of control. Communication may have the form of asynchronous message passing, implicit invocation through events, or remote procedure calls. When the Event bus pattern is implemented with independent processes or with objects with their own thread of control, this pattern is an example of this style. The Client-server pattern and Peer-to-peer pattern are examples as well.
- In the *Dataflow style*, data flows from one component to another, in a stream. A pattern that belongs to this style is the Pipe-filter pattern. Some instances of the Client-server pattern also belong to this style, for instance when a client is used to receive and display streaming audio or video sent by a server.
- The *Data centered style* means that data is stored centrally. An example of this style is the Blackboard pattern. Again, instances of the Client-server pattern may belong to this style, when the main function of the server is to manage a database, and clients are used to access that database.
- In the *Hierarchical style*, the system is partitioned into subsystems with limited interaction. Patterns within this style are the Interpreter and the Layers pattern.
- The *Call and return style* has the calling process wait for the return of request. Patterns within this style are the Master-slave and, again, the Layers pattern. Object-oriented systems without threads are also an example of this style.

4.4 Applying patterns: KWIC example

A classical example to illustrate the differences between architectural patterns is the KWIC problem: *KeyWord In Context*, introduced by Parnas [39]. In this section we introduce this problem, and show how different architectural patterns can be used to solve it. The KWIC example has been used for the first time to illustrate architectural styles in [48].

The keyword in context problem takes as input a list of lines; a line is composed of words. The output of the solution for the problem should be all "circular shifts" of all lines, sorted. A circular shift is done by repeatedly removing the last word and appending it at the beginning of the line. For example, with the input:

```
man eats dog
```

the output should be:

```
dog man eats
eats dog man
man eats dog
```

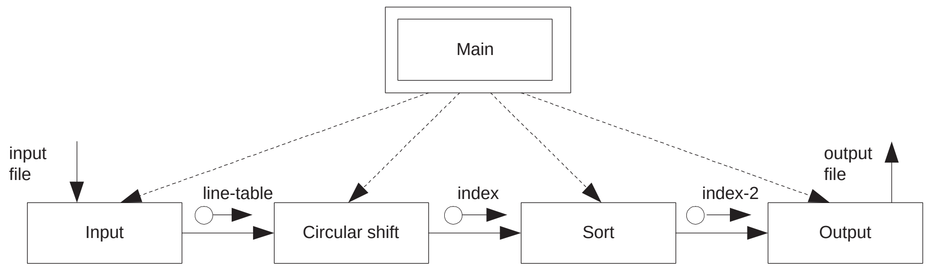

Figure 4.16: KWIC, the classical solution

4.4.1 Shared data

Traditionally, the KWIC problem is solved using shared data and functional decomposition. Figure 4.16 is a *Yourdon structure chart* [56] for the classical solution to the KWIC problem. The processing elements are subroutines, and an arrow means an invocation. The small arrows show which data is added to the shared data.

The Main subroutine invokes functions. The first invoked function reads the input file, and adds a table of lines to the shared data. The second function performs the circular shift, and adds the various shifted lines to the shared data (as a series of indexes to indicate the order). The third function then sorts all these lines, and adds the resulting lines to the shared data (once again using indexes). The last invoked function writes the result to an output file.

4.4.2 Layers pattern

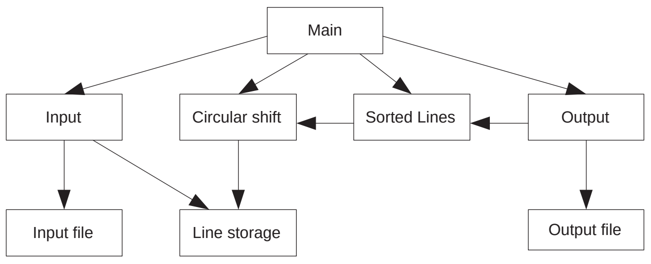

Figure 4.17: KWIC, Layers pattern

An alternate solution to the KWIC problem uses the Layers pattern. In Figure 4.17, the elements are objects, and the arrows denote method calls. No data is shared: communication of data is done through method calls only.

The advantage of this solution is that the data is hidden inside the objects, which means that choices for data structures and algorithms are encapsulated in specific components, and may therefore be more easily changed, by keeping the abstract interfaces as they are.

The organization in layers is based on objects only calling methods from objects in the same layer or in the layer directly beneath them.

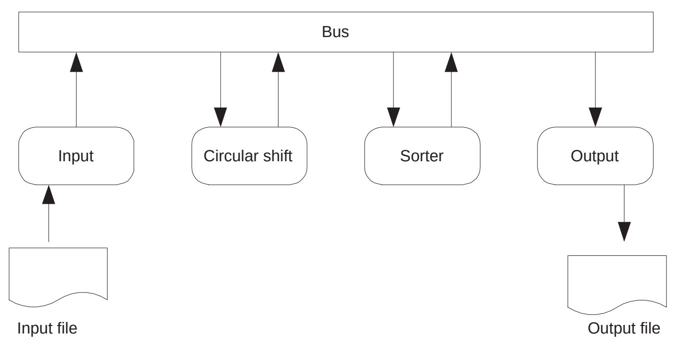

Figure 4.18: KWIC, Event bus pattern

4.4.3 Event-bus

Another solution to the KWIC problem uses the Event-bus architecture. Components that are interested in certain events are notified when those events become available. That means that components are called implicitly.

Figure 4.18 is a dataflow diagram [57]. Elements are processes, and the arrows are data in transfer between processes. The shifter will be invoked after the input process has inserted the first line. Data should be shared between the input process and the circular shift, and among the circular shift, the sorter and the output process. The sorter is invoked each time the circular shifter has inserted a line in their shared data.

This solution does not offer the advantages of data hiding of the previous solution, but it can be quicker because there may be parallelism. However, it is a complicated solution, with two shared data spaces and an event bus.

4.4.4 Pipe-filter

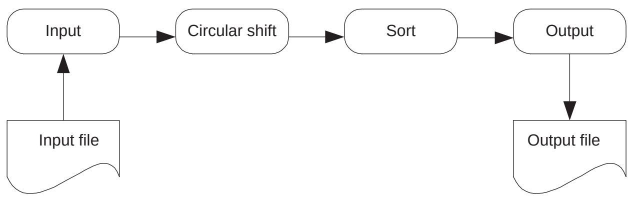

Figure 4.19: KWIC, Pipe-filter pattern

Yet another possible solution to the KWIC problem uses the Pipe-filter pattern. In this case a chain of transformers is constructed. We need a uniform data format to do that.

The same amount of parallelism as has been achieved in the event bus solution is possible here, in a less complicated system.

The same amount of data hiding as in the layered object-oriented solution is possible, with the only drawback that there is a uniform data format for the pipes. This makes the filters easier to work with, but when different data structures are chosen inside the filters, there may be overhead because of translation between different data structures.

4.5 Choosing a style

Which architectural pattern is best for a given situation depends on which requirements have the highest priority, such as:

- *Maintainability:* How easy or difficult is it to add an additional processing component, for instance to filter certain words? How easy or difficult is it to change the input format, such as adding line numbers? In the Pipe-filter pattern for instance, adding a filter is very easy, but changing the input format might be hard.
- *Reusability:* Can individual components be reused in other systems? In this case, the Pipe-and-filter pattern enhances reusability because of the uniform data format that is used.
- *Performance:* Is the response time small enough? Is overall resource behavior (memory usage in this example) acceptable? Patterns that make use of parallelism, such as the Pipe-filer pattern and the Event-bus pattern, will have better performance. On the other hand, starting a complex system like the event bus system, or transforming data in every filter using a different data structure, may lower performance.
- *Explicitness:* Is it possible to provide feedback to the user? Per stage? This is not possible in the Pipe-filter pattern for instance.
- *Fault tolerance:* For this example, there is no difference between the different solutions, but had a Master-slave pattern been applied, fault-tolerance would have been enhanced.

The list of requirements and their priorities will vary for every system. No rigid guidelines can be given to tell you which pattern will be the best in every case. Much also depends on the implementation of the pattern. Independent processes, for example, may be implemented using threads or using processes on different machines. The balance between communication and computation, the capacity of the processors involved and the speed of communication between machines, among others, will decide which implementation will have the best performance.

DISCUSSION QUESTIONS

1. In the previous module, we introduced the name tactic for an architectural decision that influences the quality of the product. Architectural patterns can be viewed as being constructed from tactics. Can you give some examples?
2. Once a pattern has been used in an architectural design, the reason for its use may be forgotten and over time the design may start to diverge from the pattern. Do you think this should be prevented? If so, how?
3. Design patterns often represent crosscutting concerns that can be implemented using aspect-oriented programming or program transformation techniques. Is the same true of architectural patterns?
4. In [8], the architecture of Linux is distilled from the source code. Do you recognize patterns or styles in this architecture?

Module 5

Patterns for enterprise applications

INTRODUCTION

Enterprise applications are software applications that perform business functions such as accounting, production scheduling, customer information tracking, bank account maintenance and the like. These applications are almost always hosted on servers, and are used by multiple employees of the same organization, and sometimes by customers as well.

Major subcategories of enterprise applications include enterprise resource planning, customer relationship management, and supply chain management. Other categories supply operations specific to the users' industry, for example banking, insurance, universities, hospital management, or civilian government.

Enterprise applications have similar architectural characteristics. In this module, we discuss these characteristics, and patterns for enterprise applications.

LEARNING OBJECTIVES
After having studied this module you should be able to:
- explain the characteristics of enterprise applications,
- describe problems in each of the three layers of enterprise applications,
- describe patterns to solve those problems,
- explain the advantages and the disadvantages of those patterns.

5.1 Enterprise applications

An enterprise application usually has the following characteristics:

- *Persistent data*: Persistent data are data that are used in multiple runs of the program. These data often persist for years and outlast programs and hardware.
- *Voluminous data*: The volume of data in enterprise applications is often in the order of millions of records, usually organized in databases.
- *Concurrent access*: Concurrent access causes the need to ensure that two people do not update the same data simultaneously. An example is formed by web-based systems: many users may access such a system simultaneously.
- *Complicated user interface*: User interfaces often consist of hundreds of distinct screens, customized for many different types of user.
- *Integration with other applications*: As an enterprise application rarely lives in isolation, it should integrate with other applications: legacy software for instance. Different technologies may be used, and different collaboration techniques, which makes integration a difficult job. Problems are even greater when integration with software of business partners is needed. Sometimes even subtly different meanings for the same fields, called *conceptual dissonance*, cause integration problems.
- *Business logic*: Business logic defines the operations, definitions and constraints that apply to an organization in achieving its goals. Business logic is represented as a set of conditions, constraints and business rules. Unfortunately, business logic often has the form of an haphazard array of strange conditions that have come about by seeking small advantages, and cannot be influenced by the IT department.

5.1.1 Examples

Our first example of an enterprise application is a *Business to Consumer* (B2C) online retailer, a business that sells goods to customers over the internet. Amazon.com, the online bookseller that launched its site in 1995, is a well-known example of a B2C online retailer. The major characteristic of this kind of application is the high volume of users. That is why scalability is a high priority requirement for such a system. An application for a B2C online retailer must be easily scalable by adding additional hardware. The user interface of such applications has the form of a web presentation: everyone should be able to access the application.

Our second example is an application to process lease agreements. A lease agreement may concern renting buildings, cars or equipment, over a long period of time, especially for business use. This kind of application serves fewer users than an application for a B2C online retailer does, has more complicated business logic (transactions may take hours, for instance), and requires sophisticated presentation, often in the form of a rich client (as opposed to a web-based user interface).

Our third example is an application for expense tracking for a small company. Such an application has just a few users. Its business logic is simple. An important requirement for such an application is that it should be easily and quickly adjustable, according to the development of the company. Reusing the architecture and the development methods of the two examples above would slow down the development of such a application.

5.1.2 Three principal layers

An architectural pattern that is usually applied to enterprise applications is the Layers pattern (see the module on architectural patterns). Enterprise applications have three principal layers:

- The *presentation logic layer* is responsible for the interaction with the user. A command-line interface may be used, but in most cases the interface is a rich client or a web interface.
- The *domain logic layer* contains the main functionality of the system. Here, input is validated and results are calculated. This layer is often structured with objects that model real-world concepts.
- *The data source logic layer* implements the communication of the domain logic layer with a database, with messaging systems or with other applications.

5.1.3 Example: Java EE architecture

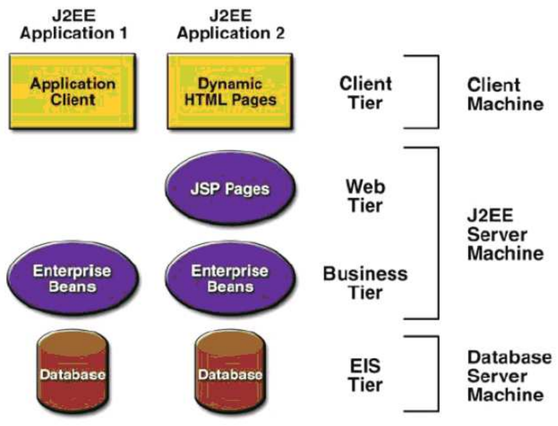

Figure 5.1: Java EE architecture

An example of an architecture with these three layers is the *Java EE multi-tier architecture*. Figure 5.1 shows four tiers on three machines. The difference between the concepts of *tier* and *layer* is that a layer is concerned with the logical separation of functionality, whereas a tier is concerned with the physical separation of functionality. In this example, the web tier and business tier may run on two different machines; in this case, they run on the same machine.

The client tier runs on the client machine, the web tier and the business tier on the Java EE server machine, and the EIS tier on the database server machine. The client tier and the web tier together form the presentation layer. The business tier contains the domain logic and the datasource logic; the EIS tier contains the data that the data source tier makes available.

The presentation logic in the Java EE architecture may be implemented using servlets and JSP (JavaServer Pages), applets, JavaScript and Java applications (rich clients); the business logic and the data source logic are implemented by Enterprise JavaBeans.

- *Servlets* contain Java code to generate HTML, similar to CGI (Common Gateway Interface) scripts. Servlets all run in the same JVM (Java Virtual Machine) which eliminates startup costs (CGI scripts run as separate processes, and thus incur startup costs).
- *JavaServer Pages* (JSP) are similar to PHP pages: JSP pages contain code embedded in HTML. Neither servlets nor JSP clearly separate HTML and logic. The logic might be restricted to variables, but often the logic needs repetition or choice as well. This results in pages with scripts and HTML entangled.

- *Enterprise JavaBeans* (EJB) are reusable components. They may be used for entities (for instance database objects) as well as for sessions (the process of communication with a client). They are inherently distributed: communication is done through the Java Remote Method Invocation (RMI) system or through CORBA.

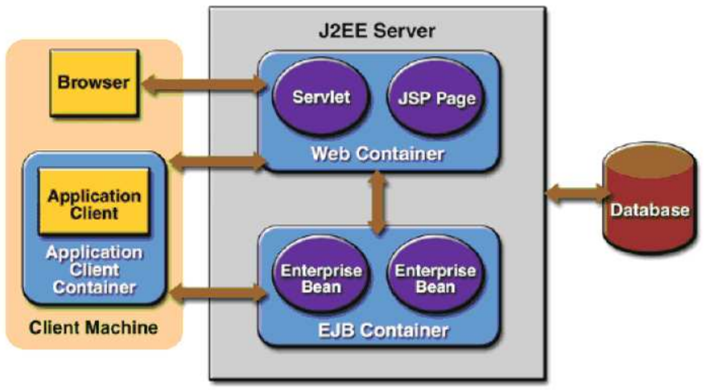

Figure 5.2: Java EE server and containers

An *EJB server* provides an environment that supports the execution of applications developed with EJB components. It manages and coordinates the allocation of resources to the applications. Enterprise beans typically contain the business logic for a Java EE application.

An EJB server (like JBoss) provides one or more EJB containers (Figure 5.2). An EJB container manages the enterprise beans contained within it. For each enterprise bean, the container is responsible for registering the component, providing a remote interface for the component, creating and destroying component instances, checking security for the component, managing the active state for the component, and coordinating distributed transactions. Optionally, the container can also manage all persistent data within the components.

5.2 Patterns in the three layers

Each layer of an enterprise application has its own problems, and patterns to solve them. We discuss the problems and the patterns by layer. The patterns are all described in detail in [19]

5.2.1 Domain logic layer

For the *domain logic layer*, we discuss the Transaction script pattern, the Domain model pattern, and the Table module pattern.

Transaction script pattern

The *Transaction script pattern* simply uses a single procedure for each action that a user may want to do, such as *Book a hotel room*. An action may be as simple as viewing some information in a particular way, or as complicated as making several changes in several databases. The advantage in enterprise applications, is that business logic is organized around procedures, and that each procedure handles a single transaction. A transaction has a well-defined endpoint, and should therefore be complete, on an all or nothing basis.

In this pattern, a transaction may be organized as a separate class for each transaction (in which case, it is an instance of the Command design pattern), or in classes that each define a subject area of related transaction scripts. Calls to the database will be made directly, or through a thin database wrapper.

The pattern is simple and easily understood, but it will inevitably lead to duplicated code. The Transaction script pattern is in fact a form of classic procedural programming, and has the disadvantage that there is a risk of duplication between transactions. Refactoring all the duplication away will eventually lead to the Domain model pattern.

Domain model pattern

A *domain model* is an object oriented class model of a domain that incorporates both behavior and data. A domain model should be independent of other layers. It is therefore usually easy to develop and test in isolation.

A domain model differs from a database model. Within a domain model, you may make use of inheritance and of design patterns. Attributes within a class may be objects themselves, and behavior is an integral part of the model. Therefore, it is hard to map a domain model to a database. Another disadvantage is the risk that data and responsibility needed for single use cases lead to a bloated model.

Table module pattern

A *table module* is a single instance that handles the business logic for all rows in a database table or view. The traditional object-oriented approach is based on objects with identity, along the lines of a domain model. Thus, if we have an Employee class, any instance of it corresponds to a particular employee.

A table module organizes domain logic with one class per table in the database, and a single instance of a class contains the various procedures that will act on the data. The primary distinction with domain models is that, if you have many orders, a domain model will have one order object per order while a table module will have one object to handle all orders.

The Domain model and Table module patterns are similar in that data and behavior are packed together; the difference is in what an instance stands for: one object with identity in the domain model pattern; all rows in a database table (translated to an object model, every row would be translated in a single object) for the table module pattern.

A mapping from this pattern to a database is very easy. Disadvantages are that direct instance-to-instance relationships are not possible. For instance, when a row within a table represents a person, relations between persons are not possible within the table module pattern. Inheritance is only possible between tables; not between rows of a table. Because the rows correspond with the objects in the domain that we model, it will, in general, not be possible to use design patterns with this pattern, in contrast to the domain model pattern. So, for complicated domain logic, the domain model pattern works better.

An example of this pattern can be found in Microsoft COM and .NET applications, where the *record set* is the primary repository of data in an application, which can be passed directly to the user interface.

5.2.2 Data source logic layer

In general, the data source logic layer has the responsibility to communicate between the object-oriented domain model layer, and a database management system that "talks" *SQL*. The issues are:

- It is awkward to embed SQL in a programming language, for instance because programmers and database experts are different people. The general rule is to keep SQL in separate classes, distinct from domain logic.
- There is a behavioral problem: the right objects should be loaded, and should save themselves when modified. This can be complicated when several related objects of the domain model are in memory (for instance a room, a person with an address, and a reservation). Somehow, you must be certain that the data for these objects are not modified without notifying the objects. In the other way around, you must make certain that the database still is consistent when the objects are modified and save their data to the database.
- The mapping of objects to tables poses a structural problem. Objects can handle multiple references, by means of collections. When you map such a relationship to tables in a database, the direction of the association is reversed (this will be discussed in more detail below).

Mapping classes to tables

As long as they are not temporary, attributes of a class may be mapped to columns in a table. In principle, each class maps to a table.

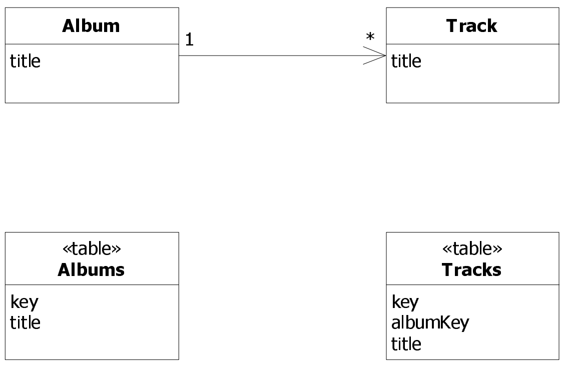

Figure 5.3: Mapping of a collection field

In Figure 5.3, class `Album` holds multiple references to objects of class `Track`. In the database, this is reversed: class `Album` is mapped to a table `Album`, with columns `key` and `title`. Class `Track` is mapped to a table with columns `key` and `title`. In the class diagram, `Album` has an attribute `tracks` that points to a collection of `Tracks`. Such an attribute cannot be mapped to a column of table `Album`. Instead, table `Track` gets a third column, `albumkey`. To see the tracks belong to a certain `Album`, we first need the `key` of that `Album`, and then we do a query on the `Track` table, to get all `Tracks` that have that specific `albumkey`.

Figure 5.4 shows two classes: `Employee` and `Skill`, related through a *many-to-many association*. The class `Employee` is mapped to a table with at least a column for the `key`. The class `Skill` is mapped to a table in the same way. To establish the relationship, we need a third table, with a `key`, an `employeekey` and a `skillkey` as columns. To

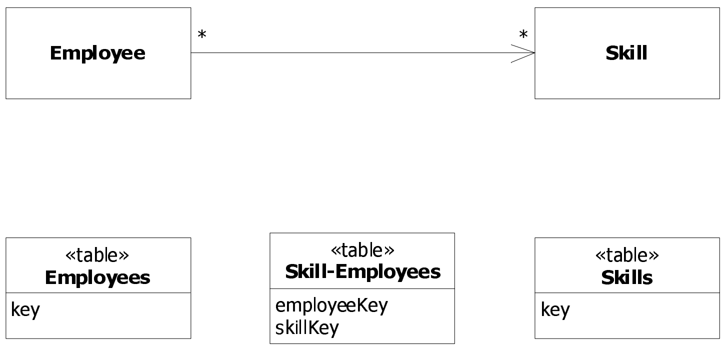

Figure 5.4: Mapping of a many-to-many association

find all skills of a certain `Employee`, we first look up the `key` of that `Employee`, then do a query on the `Skill-Employees` table to get all rows with that `employeekey`, and then we do queries on the `Skills` table to find the skills that are pointed to by the skillkeys we have found.

To find all Employees with a certain skill, we do the same, the other way around. This is the *association table mapping pattern*.

Record set pattern

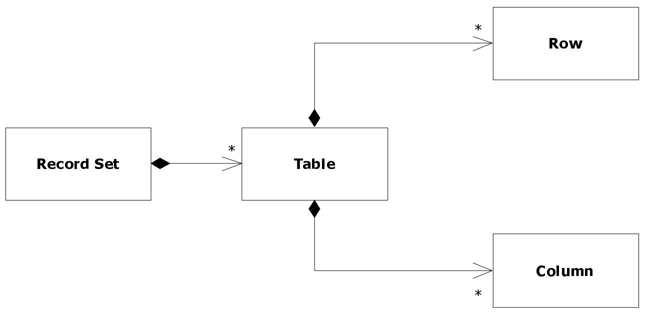

Figure 5.5: Record Set pattern

Within the *Record set pattern* (Figure 5.5), you use an in-memory representation of tabular data. In general, record set objects are provided by your software platform (for instance, the *row sets* of JDBC). To get data from this, you usually have to invoke a generic method with a (String) argument to indicate which field you want. Record sets are easily manipulated by domain logic when you use the table module pattern.

Table data gateway pattern

A *gateway* in general is an object that encapsulates access to an external system or resource (see Figure 5.6). A gateway in the data source layer thus wraps all the special API code (such as from JDBC, or SQL) in a class. The advantage of gateways is that other objects do not need to be aware of the API or of a special language like SQL.

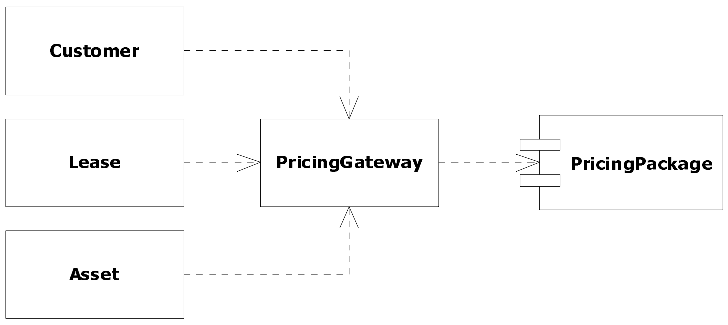

Figure 5.6: Gateway pattern

A *table gateway* is a gateway that holds all the code for accessing a single table or view. The table gateway may return the information in different forms:

- in a single data structure (for instance a Map). The disadvantage is that this does not allow compile time checking.
- in a record set, as provided by the platform (see the record set pattern above). This works well with the table module pattern in the domain logic layer.
- in a domain object from the domain model. This solution forces bidirectional dependency, which is a disadvantage, because either the domain model will be structured as the database, or the other way around. The Data mapper pattern (discussed below) provides a better solution for the domain model pattern.

Data mapper pattern

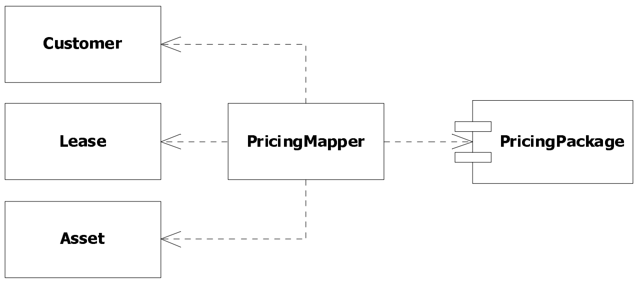

Figure 5.7: Mapper pattern

In general, a *mapper* is an object that sets up a communication between two subsystems that should stay ignorant of each other, and should also be unaware of the existence of the mapper, for instance between architectural layers. A mapper may be invoked by a third subsystem, or as an Observer. See Figure 5.7.

A *data mapper* (Figure 5.8) is a layer of software that separates the in-memory objects from the database. The data mapper encapsulates decisions about translation between domain model and database schema. It introduces an extra layer between the domain logic and data source logic layers.

With a data mapper, the in-memory objects (from the domain model) need not be aware that there is a database present (and the database never knows about the objects that

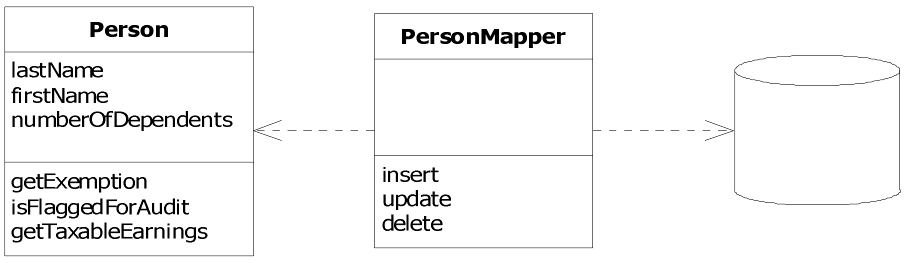

Figure 5.8: Data mapper pattern

use it), which is an advantage. As Figure 5.9 shows, the mapper first checks if the domain object is already loaded. If not, it accesses the database, extracts the relevant information, and creates the desired object. A problem is that objects are often interrelated; you have to stop pulling data back at some point. In this case, you may use the Lazy load pattern, that does just what the name suggests.

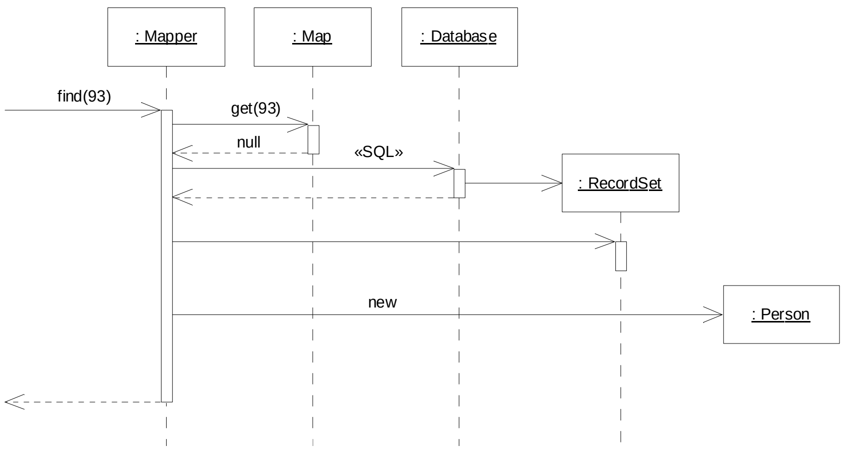

Figure 5.9: Sequence diagram for the data mapper pattern

5.2.3 Presentation layer, server-side

Web-browser-based interfaces have many advantages:

- No specialized client software has to be installed: the browser is used as a platform for the web presentation.
- The application is accessible from anywhere.
- User interfaces follow a common approach.

What a web server does is interpret the URL of a request and hand over control to an appropriate program or serve a static HTML page. Two kinds of programs are distinguished, as we will discuss: scripts and server pages. A single web server can handle many kinds of programs. A configuration file indicates which URL is to be handled by which program.

For the presentation layer, this provides two ways to use logic (as opposed to using static HTML): scripts and server pages.

Server-side scripts

Scripts are programs in a general high-level programming language, that are interpreted on the server. A script may get data from parsing the HTTP request. It will do some logic operations (often involving the logic layer), and generates HTML that is sent back to the client.

An example of scripts on the server is formed by *Common Gateway Interface* (CGI) scripts. They can be written in various languages. Perl is a popular language for CGI scripts, because of the ease of regular-expression parsing. Another example is formed by Java *servlets*. Servlets run as threads in one JVM, as we have said. Parsing of the HTTP request is automated. The output (HTML) is generated with normal output stream operators, which means that coding servlets is easy for any Java programmer. However, web presentation is often the responsibility of graphical designers, and they do not always have the programming skills to construct servlets (or other scripts).

As a side note, server scripts are not to be confused with client-side scripts, that may be embedded in an HTML page. We will discuss those later.

Server pages

```
<html>
 <body>
  <%! Subjects s = new Subjects(); %>
  Click on any of these subjects:
  <%
   Iterator all = s.getAll();
   while (all.hasNext())
   { String Url = (String) all.next();
  %>
  <p>
   <a href = "http://<%=␣Url%>.jsp">
         <%= Url%>
   </a>
  </p>
  <%}%>
 </body>
</html>
```

Figure 5.10: Server page

Server pages are pages written in HTML interspersed with code fragments, *scriptlets* (see Figure 5.10). These code fragments are interpreted on the server, and will output something that fits in the HTML code around it.

The advantage over servlets is that it is easier to see how the resulting HTML page will look. The server pages technique works best if the results involved need little processing. Examples of server pages are PHP (Hypertext Preprocessor), ASP (Active server pages) and JSP (Java server pages).

In both techniques (scripts and server pages), logic and view (in the form of HTML) tends to get tangled: in scripts, you have to follow the print-statements to see what HTML is produced, whereas in server pages, you have to follow the scriptlets to see what logic is involved. Several patterns help tackle this problem in the presentation layer.

Model-view-controller pattern

This is the same pattern that has been discussed in the module about architectural patterns. Here it is applied to web presentation.

- The *model* represents information about the domain. In enterprise applications, the logic layer is responsible for the model, usually using a domain model.
- The *view* is a display of the model in the user interface. In web presentations, the view is an HTML page, generated or static.
- The word *controller* has many interpretations, which has lead to many misunderstandings of the Model-View-Controller pattern. It would be better to use the word *input controller*: the controller takes input, forwards it to the model, which may change as a reaction, and causes the view to be updated.

In standalone applications, the separation of view and controller is usually neglected. In web applications, it is useful to use a script (a servlet for instance) for input interpretation, and a server page for response formatting.

Page controller

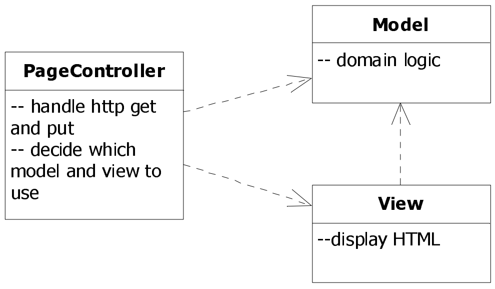
Model
-- domain logic
PageController
-- handle http get and put
-- decide which model and view to use
View
--display HTML

Figure 5.11: Page controller pattern

A *page controller* is an object that handles an action initiated on a specific web site, with one controller per page. The web server receives HTTP requests (both GET and PUT requests: the difference is that GET requests consist of a URL, sometimes followed by parameters, whereas a PUT request consists of a URL and a separate 'package' of data), and passes control to the page controller. The page controller inspects the URL and the data, and decides what to do: data will be carried over to the model, which will process them, and the results will be passed to a script or server page in the view, that will show the results in a page. An alternative to page controller is a *front controller*, which handles all requests for an entire web site.

As in the MVC pattern, the page controller is often a servlet, although the view is implemented by server pages. The responsibilities of the page controller are:

- Decode the HTTP request and extract (form) data.
- Create and invoke model objects to process the data.
- Notify the appropriate view of the resultant changes in the model.

Figure 5.12 shows a servlet functioning as a page controller. The method

```
doGet(HttpServletRequest, HttpServletResponse)
```

is a standard method in `javax.servlet.http.HttpServlet`, that one has to redefine to get the required behavior. The method forward is not standard, but is defined

```
class ArtistController
  extends javax.servlet.http.HttpServlet
{ ...
  public void doGet
    (HttpServletRequest request, HttpServletResponse response)
  throws IOException, ServletException
  { Artist artist =
      Artist.findNamed(request.getParameter("name"));
    if (artist == null)
      forward("/MissingArtistError.jsp", request, response);
    else
    { request.setAttribute
        ("helper", new ArtistHelper(artist));
      forward("/artist.jsp", request, response);
    }
  }
 }
}
```

Figure 5.12: Servlet as page controller

in a superclass for all page controllers. It is supposed to forward the result to the appropriate view to display the result. The server page `artist.jsp` is an example of a template view with a helper object, which will be explained below.

Template view pattern

```
class ArtistHelper
{
  private Artist artist;

  ...

  public String getAlbumList()
  {
    StringBuffer result = new StringBuffer();
    result.append("<ul>");
    for (Iterator i = artist.getAlbums().iterator();i.hasNext();)
    {Album album = (Album) i.next();
      result.append("<li>");
      result.append(album.getTitle());
      result.append("</li>");
    }
    result.append("</ul>");
  }
 }

 <%= helper.getAlbumList()%>
```

Figure 5.13: Template view with helper object

```
<album>
  <title>
    Rubber Soul
  </title>
  <artist>
    The Beatles
  </artist>
  <tracks>
    <track>
      <title>
        Drive My Car
      </title>
    </track>
    <track>
      <title>
        Norwegian Wood
      </title>
    </track>
    ...
  </tracks>
</album>
```

```
<xsl:template match="album">
 <html>
  <body bgcolor="white">
   <xsl:apply-templates/>
  </body>
 <html>
</xsl:template>
<xsl:template match="album/title">
 <h1>
  <xsl:apply-templates/>
 </h1>
</xsl:template>
<xsl:template match="artist">
 <p>
  <b>Artist: </b>
  <xsl:apply-templates/>
 </p>
</xsl:template>
```

Figure 5.14: Transform view

A *template view* renders information in HTML by embedding markers in an HTML page, that are replaced at view time by the result of a computation. The Template view pattern can be implemented in different ways:

– Using *XML tags*, which are ignored by WYSIWYG (what you see is what you get) HTML editors. Graphical designers can therefore use template pages in such an editor, without any programming experience.
– Using *special text markers*, which give you more syntactical freedom than markers in XML.
– Using *server pages*, such as ASP, PHP or JSP. Server pages are an instance of the Template view pattern. The advantage is that it is possible to represent arbitrary program logic, but this is a disadvantage as well, because the separation between model and view is easily broken, and because graphical designers (responsible for the look and structure of the page) and programmers (responsible for the logic) have to work on one and the same page.
– Using *helper objects* that contain programming logic. In this implementation you define a helper object for every page, and let scriptlets on the page only make calls to this helper object. It is then easier for non-programmers to maintain the page. In Figure 5.13, the class ArtistHelper is the class for the helper object; the scriptlet on the page where the helper object is called is shown below the source of the class.

Using helper objects is much cleaner than embedding the repetition in the server page as shown in Figure 5.10. With helper objects, the server page only has to contain a single method call (as a Java expression, not a scattering of code fragments). The helper class does contain a little HTML, but only for formatting purposes.

Transform view pattern

In the *Transform view pattern*, domain data is processed element by element, and transformed into HTML. Where a template view is organized around the form of the output, a transform view is organized around the separate transforms for different elements.

In Figure 5.14, every time a tag `album` is encountered in the xml snippet on the left, HTML tags for html and body are written as is specified in the xslt snippet on the right, the transforming process goes on, and when done, the end tags are written. Every time a tag `title` is encountered within an album, a `H1` HTML tag is written, the title is written, and the `H1` endtag is written.

In implementations of this pattern, the domain data is most often encoded in XML, and the transformation is most often done using *XSLT* (Extensible Stylesheet Language Transformation), a functional language for presenting XML data in HTML form, or any other form. Several API's have the possibility to present the result of the domain logic as XML, e.g. JAXB. The XML is domain oriented; in the XSLT we decide the form the output takes as a web page.

The Transform view pattern avoids two of the biggest problems with template view: The transform is focused on rendering HTML and avoids mingling this with domain logic, as is often the case within the Template view pattern. Furthermore, it is easy to run the transform view and capture the output for testing, while the template view can only be tested on a web server.

5.2.4 Presentation layer, client-side

There are several techniques to provide a richer user experience than is possible with plain (static or server-generated) HTML.

Java applets

Java applets are Java programs that run within the browser. The advantages are that they may provide the look and feel of a rich client within the browser, and that you have the full power of Java. Security issues are solved: applets run within a sandbox, imposing strict limitations on access to system resources.

Disadvantages are that applets require the browser to have the right version of Java, and that there is no cooperation from e.g. Microsoft Internet Explorer: users must then install Java separately to be able to make use of applets.

Canvas

The HTML5 *Canvas* element makes it possible to draw, using a script language such as JavaScript, in that element on the webpage.It is possible to add events handlers to areas in the drawing, which makes it possible to create graphical applications with interaction.

Advantages are that it makes highly dynamic pages possible, that no plugin is required, and that it works with every modern browser.

A disadvantage is that it needs JavaScript programmer skills to develop an application. There are libraries available, but one needs to be an experienced JavaScript programmer to work with the HTML5 canvas element. An authoring environment for non-programmers (still) lacks.

DHTML

DHTML stands for *Dynamic HTML*. It consists of the combination of HTML, CSS, JavaScript and the DOM.

- *HTML* is the markup language for web pages.
- *CSS* (Cascading Style Sheets) makes it possible to inform the browser about styles (color, font, width of elements, etc) to use for HTML elements.
- *DOM* stands for Document Object Model. Using the DOM, a web page is represented as an object, and each element of the page can be accessed separately. Accessing elements of a page can be done using client-side scripting, for instance using JavaScript.

Using DHTML, it is possible to dynamically change style properties of a certain element of a page (for instance by making elements visible or invisible). Such changes may be done as a reaction to input from the user (from the mouse or the keyboard) or changes may be timed. Because the scripts are interpreted by the browser, there is no need to communicate with the web server. Using the canvas element to create graphical, interactive programs, is a specialized example of DHTML.

The advantage of DHTML is that it requires no plugin, and that it works within every (new) browser (unless the user has disabled JavaScript). Disadvantages are that older browsers may support different versions of JavaScript, that different browsers have different interpretations of the Document Object Model, and may also differ in their support for CSS. Support is getting better with each new version of browsers, fortunately.

XML derivatives

Several *XML derivatives* exist, such as:

- XUL (XML User Interface Language), for Firefox,
- XAML (eXtensible Application Markup Language), for Microsoft Windows Vista,
- MXML (Maximum eXperience Markup Language), for Macromedia Flex + ActionScript,
- XAMJ: open-source alternative to XAML, based on Java (clientlet architecture).

The disadvantages are obvious: at this moment, there is no standardization between browsers.

Ajax

Ajax is an acronym for Asynchronous JavaScript And XML. It was first introduced by James Garrett of Adaptive Path in 2005 [21]. Ajax is based on the `XMLHttpRequest object`, which is not standardized, but implemented similarly in Internet Explorer, Firefox, Mozilla, Safari, Opera, Konqueror, etc.

The general idea of Ajax is to enhance the responsiveness of web pages. Instead of getting a whole new page after some action of the user (with the corresponding wait time), an `XMLHttpRequest` is sent to the server. The server responds by sending back data (for instance encoded in XML), which are placed on the page by JavaScript (by accessing the DOM), without a page reload.

Advantages are that it requires no plugin, only JavaScript in the browser, and that it works with arbitrary server-side techniques. Disadvantages are that it is easy to violate user's expectations about the behavior of web applications. It is possible to implement the functionality of the "Back" key for instance, but the developer has to implement it explicitly. Many Ajax sites therefore do not react to the Back key as the user expects.

Well-known examples of the use of Ajax are Google's applications (GMail, Google maps), Flickr or Amazon.

5.3 Concurrency problems

Enterprise applications are inherently concurrent. Concurrent applications are almost impossible to test, and hard to reason about because of state explosion. Most concurrency problems in enterprise application development can be avoided by using transactions.

An even harder problem is formed by offline concurrency, because interactions cannot be placed within a single database transaction. An example containing this kind of transactions is version control systems.

This section discusses several patterns to tackle concurrency problems.

5.3.1 Optimistic and pessimistic locking

Using *optimistic locking*, all users can freely read and edit the file or resource. Only at the point of committing changes, concurrency control detects conflicts. This pattern works if conflicts are relatively rare and not very painful (for instance when automatic merges are possible).

Pessimistic locking means that whoever checks out the file or resource first prevents anyone from editing it. This pattern prevents conflicts rather than detecting them. A disadvantage is that it hinders progress. Another disadvantage is that it may lead to a deadlock situation (imagine two eaters, one with a fork and one with a knife, each waiting for the other utensil before being able to eat their stake).

You should always pick optimistic locking, unless the consequences of a conflict are completely unacceptable.

5.3.2 Transactions

A transaction is a bounded sequence of work with well-defined begin and end points. Software transactions should be ACID: atomic, consistent, isolated and durable.

– *Atomic* means that every step must complete successfully within the bounds of the transaction or all work must roll back.
– *Consistent* means that the system must be in a non-corrupt state between transactions.
– The transactions are *isolated* when the results of a transaction in progress are not visible until it commits successfully.
– The transactions are *durable* when the results of a committed transaction are permanent, i.e. survive a crash of any sort.

The SQL standard defines several levels of transaction isolation. Several problems may occur in concurrent transactions. The highest level of isolation avoids them all; the lowest level of isolation allows them all. The problems that may occur are:

– *Dirty Read*: A session reads rows changed by transactions in other sessions that have not been committed. If the other session then rolls back its transaction, subsequent reads of the same rows will find column values returned to previous values, deleted rows reappearing and rows inserted by the other transaction missing.
– *Non-repeatable Read*: A session reads a row in a transaction. Another session then changes the row (Update or Delete) and commits its transaction. If the first session subsequently re-reads the row in the same transaction, it will see the change.
– *Phantoms*: A session reads a set of rows in a transaction that satisfies a search condition (which might be all rows). Another session then generates a row (Insert) that satisfies the search condition and commits its transaction. If the first session subsequently repeats the search in the same transaction, it will see the new row.

The *levels of isolation* are:

- *Serializable* transactions completely isolate each transaction from other active transactions. After initiation, a transaction can only see changes to the database made by transactions committed prior to starting the new transaction. This is the highest level of isolation. Serializable transactions are necessary to be sure of correctness, but they are very bad for system liveness and throughput.
- *Repeatable Read* transactions allow for phantoms.
- *Read committed* transactions allow for unrepeatable reads.
- *Read uncommitted* transactions allow for dirty reads.

5.3.3 Session state

Many client interactions are inherently stateful. For example, in a shopping cart application a basket of a client is his local state. A stateful server is usually inefficient, as it needs to remember state during the complete user's session and this requires at least one object per user.

Session state is different from persistent data stored in the database. The problem with session state is isolation: it exists within a single transaction and may be inconsistent (e.g. during editing).

Client session state pattern

The *Client session state pattern* stores the session state on the client. This is easy within a rich client application. For HTML interfaces, three common ways to store the session state are:

- *URL parameters*: The session state is kept as parameters in the URL, which means that the server receives the session state with each HTTP-request. Links on a respond page take the session state as a parameter. Such a URL looks like:

  ```
  http://www.barnesandnoble.com/index.asp?userid=uF7IWlCAnh
  ```

- *Hidden fields*: Hidden fields in a form are similar to text fields, with the difference that they not show on the page. Information can be stored in a hidden field on the page, and will be sent back to the server together with the information that the user has entered in the form. A disadvantage is that they are not secure: anyone can look at the data by looking at the source of the page.
- *Cookies*: A cookie is a message given to a web browser by a web server. The browser stores the message in a text file. The message is sent back to the server each time the browser requests a page from the server. The main purpose of cookies is to identify users and possibly prepare customized web pages for them. When you enter a web site using cookies, you may be asked to fill out a form providing such information as your name and interests. This information is packaged into a cookie and sent to your web browser which stores it for later use. The next time you go to the same web site, your browser will send the cookie to the web server.

The name cookie derives from UNIX objects called magic cookies. These are tokens that are attached to a user or program and change depending on the areas entered by the user or program. Both URL's and cookies have size limitations.

5.3.4 Distribution strategies

It is not recommended to build a distributed application by putting different components of for instance the domain logic layer (such as a component for invoices, a com-

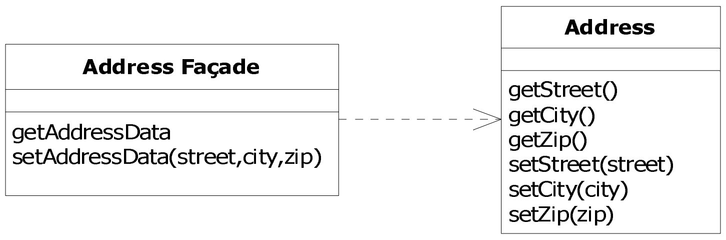

Figure 5.15: Remote Façade

ponent for customers, a component for orders and a component for deliveries) on different machines, because remote procedure calls are orders of magnitude slower then local procedure calls.

A local interface must be fine-grained to take advantage of object oriented principles like overriding and delegation. A remote interface is optimized not for flexibility and extensibility but for minimizing the number of calls that are necessary. Therefore you cannot take a component designed to work as a single process and put it on a remote machine. One effective use of multiple processors is *clustering*: run multiple copies of an entire subsystem on various nodes. Let the subsystem have a course-grained façade and minimal communication with the rest of the system. That is what the Remote Façade pattern does.

Remote façade pattern

The *Remote façade pattern* is an instance of the Façade design pattern. It provides a course-grained façade on fine-grained objects to improve efficiency over a network. The fine-grained objects contain the domain logic. Remote façade replaces all the getters and setters of the individual objects by a bulk access method, as is shown in Figure 5.15.

In Figure 5.15, the remote façade corresponds to a single domain object. More often, the façade combines data from several different objects. You may have different façades for different clients. The façade's methods form a natural point to enforce security. Otherwise, it's a thin skin with no responsibility or domain logic of its own.

DISCUSSION QUESTIONS

1. How would you decide among the various patterns handling a similar problem, e.g. among the Transaction script pattern, the Domain model pattern and the Table module pattern?
2. Do you favor server-side presentation (as in the Template view pattern) or client-side presentation (as in Ajax)? What arguments could decide the choice?
3. Should a source code repository use optimistic or pessimistic locking? Why?

Module 6

Patterns for enterprise application integration

INTRODUCTION

Enterprises are typically comprised of hundreds of applications. Those applications are different in several respects: some of them are custom built, some applications are built by third parties, and some applications are legacy software: software that has been built a long time ago, which functions and is vital to the enterprise, but often runs on old hardware while few system engineers know enough of those systems to apply changes.

Those applications of different origin should work together to fulfill the business needs: even simple customer transactions can easily span several different systems: the customer relations system, the application that monitors the inventory, the system that processes orders, applications for shipping, tax, billing, etc.

To create a single, big application to run a complete business is next to impossible. Even if it would be possible, it is not desirable: it is difficult to apply a change in such a huge system. Spreading business function across multiple applications provides the business with the flexibility to select the 'best' package for each functionality.

In this module, we will discuss concepts that play a role in system integration such as loose coupling, and discuss solutions for enterprise integration in the form of patterns.

LEARNING OBJECTIVES

After having studied this module you should be able to:

- explain the requirements for the integration of enterprise applications,
- describe different patterns to integrate applications while introducing or maintaining loose coupling,
- discuss the advantages and disadvantages of patterns for enterprise application integration.

6.1 Enterprise application integration

Simple customer transactions in enterprise applications can easily span several different systems. So, in order to support an enterprise, systems integration is unavoidable. However, all integration solutions have to deal with a few fundamental challenges:

- Integration requires a shift in corporate politics: groups will no longer control a specific application because each application is now part of an overall flow of integrated applications and services.
- Because of wide scope, integration efforts have far-reaching implications on the business: the proper functioning of the integration solution becomes vital to the business. A misbehaving integration solution can cost a business millions of dollars in lost of orders, misrouted payments, and disgruntled customers [24]. Developers of integration solutions have limited amount of control over participating applications. In most cases the applications are the legacy systems or packaged applications.
- Despite the widespread need for integration solution, only few standards have been established in this domain: examples are CORBA, XML, and web services. There is almost always a lack of interoperability between "standards compliant" products.
- Common standards of presentation do not imply common semantics. Existing XML web services standards address only a fraction of the integration challenges. The frequent claim that XML is the lingua franca of system integration is somewhat misleading. XML is like an alphabet, it is used to represent many languages which cannot be readily understood by all users. XML does no imply common semantics. The notion of account, for instance, can have many different connotations, constraints, and assumptions.
- All integration solutions have complex operation and maintenance because of mix of technologies.

6.1.1 Types of integration

Integration means connecting computers systems, companies or people. The following six types of integration appear in many integration projects.

1. *Information portals* aggregate information from multiple sources into a single display. Many business users have to access more than one system to answer a specific question. Simple information portals divide the screen into multiple zones. Sophisticated information portals provide interaction between zones so that changes in one zone affect another.
2. *Data replication*. Many systems require access to the same information. For example, a customers address may be used in the customer care system (when the customer calls to change it) and in the shipping system (to label the shipment). Both systems have their own data stores to store the customer related information. When a customer calls a change his address, all these systems need to change their copy of the customers address. Data replication means distributing changes across the data stores of all systems.
3. *Shared business functions*. Applications tend to implement redundant functionality. For example, multiple systems may need to check whether the address matches the specified postal code. It makes sense to expose these functions as a

shared business function that is implemented once and available as a service to other systems. This approach eliminates the need for data replication by allowing various systems to request data when it is needed rather than permanently save a redundant copy. Whether this is practicable depends on the amount of control we have over the systems, and the frequency of access vs. change.

4. *Service-oriented architectures*. A service is a well-defined function that is universally available and responds to requests (e.g. a shared business function). If there are many such services, we need a service directory. And each service needs to describe its interface in such a way that an application can negotiate with it. Service-oriented architectures will be discussed in a separate module.

5. *Distributed business processes* is a single distributed application. A distributed process usually contains a component to coordinate business processes. The boundary between distributed business procedures and service-oriented architecture is a fuzzy one. For example, we can expose all relevant business functions as services and then encode the business process inside an application.

6. *Business-to-business integration*. In many cases, business functions may be available from outside suppliers or business partners. Business-to-business integration comes across the enterprise boundary.

Many integration projects consist of combination of multiple types of integration.

6.2 Loose and tight coupling

The core principle behind *loose coupling* is reducing the assumptions the two parties make when they exchange information. The parties are components, applications, services, programs and users.

The opposite of loose coupling is *tight coupling*. The more assumptions two parties make about each other and the common protocol, the more efficient the communication can be, but the less tolerant the solution is of interruptions or changes.

An example of tight coupling is a *local method invocation*. Invoking a local method inside an application is based on a lot of assumptions between the called and the calling routine. Both methods have to run in the same process, be written in same language (or at least use a common intermediate language or byte code. The called method starts processing immediately after the calling method makes the call. Another example is *synchronous invocation*. In this case the calling method will resume processing only when the called method completes.

Many integration approaches make remote communications by packaging a remote data exchange into the same semantics as a local method call. This strategy resulted in the notion of *Remote method invocation* (RMI) or *Remote procedure call* (RPC). RPC or RMI is supported by many popular frameworks and platforms: CORBA, Java RMI and RPC-style web services. RMI mimics local method invocation because developers are familiar with that, and it allows postponing distribution decisions until deployment time.

6.2.1 Example: TCP/IP low-grade integration

An example of low-grade integration using TCP/IP is shown in Figure 6.1. The example is written in *C#*, but it is similar in Java or C. This code opens a socket connection to the address `securis.cs.uu.nl` and sends two items (the amount of bytes and the receiver name) across the network. No expensive middleware is required.

There are, however, some problems with this integration attempt:

```
String hostName = "securis.cs.uu.nl";
int port = 80;
IPHostEntry hostInfo = Dns.GetHostByName(hostName);
IPAddress address = hostInfo.AddressList[0];
IPEndPoint endpoint = new IPEndPoint(address, port);
Sock socket = new Socket
( address.AddressFamily},
SocketType.Stream},
ProtocolType.tcp
);
socket.Connect(endpoint);
byte [ ] amount = BitConverter.GetBytes(1000);
byte [ ] name = Encoding.ASCII.GetBytes("SWA");
int bytesSent = socket.Send(amount);
bytesSent = socket.Send(name);
socket.Close();
```

Figure 6.1: Low-grade integration: TCP/IP

- The integration method works only for byte streams: that is why we need the `BitConverter` class. The internal representation of a data type varies across systems (an integer may be 32 bits in one programming platform, but may be 16 or 64 in other systems).
- The method disregards differences in binary representation: e.g. big-endian versus little-endian formats. A big-endian format stores numbers starting with the highest byte first, while little-endian systems store the lowest byte first.
- The target computer is hard-coded. But what if we want to move to a different computer? What if the computer fails?
- TCP/IP protocol establishes temporal dependency between the machines. TCP/IP is connection-oriented protocol. Before any data can be transformed, a connection has to be established first. Establishing a TCP connection involves IP packages traveling back and forth between sender and receiver. This requires that both machines and the network must be available at the same time. If any of the three pieces are not available due to high load, the data cannot be sent.
- The protocol relies on a strict data format. In the example we send two parameters: the amount of bytes and the receiver's name. We cannot add more parameters without modifying both sender and receiver.

6.2.2 Introducing loose coupling

To move from tight coupling (like in the example of the socket connection) toward loose coupling, one should:

– remove assumptions about the data format by introducing a standard format like XML,
– remove assumptions about the address of the target machine (in the example, the hostname and the port number) by sending to an addressable channel (a logical address that both sender and receiver can agree on without being aware of each other's identity),
– remove assumptions about time by allowing a channel to queue messages,

- remove assumptions about data format by allowing for transformation inside the channel,
- and when the strategies above result in code that is becoming too complex, introduce message-oriented middleware.

6.2.3 Application integration criteria

What is a good application integration solution? The following are some main decision criteria:
- *Application coupling*. Integrated applications should minimize dependencies on each other so that each can evolve without causing problems for the others.
- *Intrusiveness*. Developers should minimize both changes to the application necessary to integrate it into the enterprise and the amount of integration code needed.
- *Technology selection*. Different integration techniques require various amount of specialized software and hardware. Such tools can be expensive, cause vendor lock-in, and increase the learning curve. On the other hand creating an integration solution without tools may mean reinventing the wheel.
- *Data format*. Integrated applications must agree on the format of the data they exchange. If existing applications do not agree on the data format and insist on different data formats, an intermediate translator is needed for unification.
- *Data timeliness*. Integration should minimize the length of time between when one application decides to share some data and other application have that data: the longer sharing takes, the greater the opportunity for applications to get out of synchronization.
- *Functionality sharing* may provide a better abstraction than data sharing. But invoking functionality in a remote application is different from local calls.
- *Remote communication*. Calling a remote sub-procedure is much slower than a local one, so a procedure may not want to wait for the result but prefer to invoke asynchronously.
- *Reliability*. Remote connections are not only slow, but much less reliable than a local function call. When a procedure calls a sub-procedure inside a single application, it is a given that the sub-procedure is available. It is not necessarily true when communicating remotely: the remote application may not be running, network may be unavailable etc.

All these different criteria must be considered when choosing an integration approach.

6.3 Integration patterns

There is no one integration approach that addresses all criteria equally well. The various approaches can be summed up in four *integration patterns* or *styles*:
- *File transfer.* Each application produces files of shared data for others to consume.
- *Shared database.* The applications store the data they wish to share in a common database.
- *Remote procedure invocation.* Each application exposes some of its procedures for remote invocation, to initiate behavior and exchange data.
- *Messaging.* Each application connects to a common messaging system.

All four integration patterns are discussed subsequently.

6.3.1 File transfer pattern

The *File transfer pattern* is applied if an enterprise has multiple applications that are being built independently, with different languages and platforms. Figure 6.2 shows a data flow diagram of this pattern (with a "file" symbol borrowed from flow charting, not UML). When the pattern is used, then:

- Each application produces files that contain the information the others must consume;
- Integrators take the responsibility of transforming files into different formats;
- The applications produce files at regular intervals according to the nature of the business.

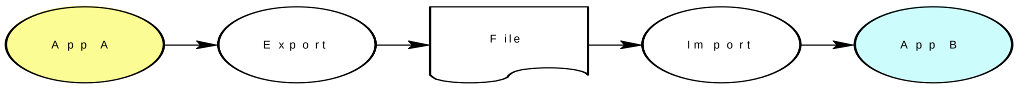

Figure 6.2: File transfer

When using the File transfer pattern, the developers must agree on, and provide

- File-naming conventions;
- Directories for file storage;
- A strategy to keep file names unique;
- The responsibility for deleting old files;
- Locking mechanism.

A component of this integration pattern gets the responsibility for deletion of old files. It must also be informed by the sender side when a file is old. Both sides must agree beforehand on the criterion.

The pattern implements a locking mechanism to ensure that one application is not trying to read the file while another is still writing it. The locking mechanism may be replaced by timing conventions.

With more frequent updates, the File transfer pattern resembles the Messaging pattern (discussed in Section 6.3.4). The remaining problem is managing all the files that get produced, ensuring that they are all read and that none of them get lost. Also processing files tends to have a high overhead, so it is better to have specialized support.

Advantages

The advantages of this pattern are the following:

- integrators need no knowledge of the internals of an application;
- no tools or integration packages are needed.

Disadvantages

The systems integrated using the File transfer pattern can easily get out of sync. Sometimes this is not a big deal: if you change your address, you expect mail to arrive at the old address for some days or even weeks. But sometimes it is really important (think of airline reservations), and in that case file transfer is not the obvious choice. A strategy must be chosen for resolving inconsistent changes.

Another example, a customer management system can process a change of address and produce an extract file each night, but the billing system may send the bill to an old address on the same day. This can lead to inconsistencies that are difficult to resolve.

Organizations should have strategies for resolving inconsistent changes. The longer the period between file transfers, the more likely the inconsistencies can become. The File transfer pattern is not suitable when everyone must have the latest data in an enterprise. Even with rapid updates of files, the inconsistencies remain a problem of the File transfer pattern.

The File transfer pattern may not enforce data format sufficiently. Many integration problems come from incompatible ways of looking at the data. This causes the so called *semantic dissonance* that cannot be resolved by data format transformations. Semantic dissonance means that different applications have similar concepts with subtly different meanings.

For instance, a "CS major" may mean:

– a student whose present major is CS (Computer Science),
– the above or a former student whose major at the time of graduation was CS,
– a student whose major at the time of initial registration was CS,
– one of the above with the proviso that their studies are financially supported by the government (no second degrees, no non-EU students),
– one of the above with the proviso that they have no outstanding financial obligations,
– one of the above with the proviso that they have not been advised to abandon the field,

All of the above mentioned semantics are actually used in different management applications within the Utrecht University!

Both problems of the File transfer pattern are solved with a central, agreed-upon data store that all of the applications share. Inconsistencies, however, remain a problem. They can be solved by assigning to one application the ownership and responsibility for a certain piece of data, but then the system will have to remember and enforce this.

6.3.2 Shared database pattern

The *Shared database pattern* (Figure 6.3) improves upon the File transfer pattern because it makes data available more quickly and enforces an agreed-upon set of data formats.

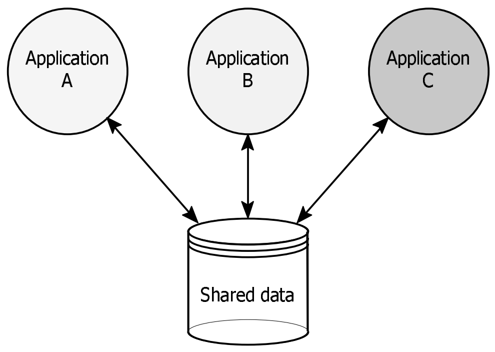

Figure 6.3: Shared database

Advantages

The Shared database pattern guarantees consistency of data. Moreover, in this pattern the simultaneous updates can be prevented by transaction management systems. Transformation of data is unnecessary. Practically all application development platforms can work with SQL. Semantic dissonance is avoided. Meaning of data is dealt with beforehand in database design.

Disadvantages

It is difficult to come up with a database design that can meet the needs of multiple applications. Often this results in a schema that individual application programmers find difficult to work with. This creates political pressure from departments to separate their data.

A harder problem is external packages. Most packaged application won't work with a schema other than their own. Also, software vendors tend to change the schema with every new release.

The database can become a performance bottleneck. A performance bottleneck occurs when many applications need to operate on the database and block each other out. It might even lead to a deadlock. Accessing a single-location database might be too slow, and a distributed database is more prone to ownership confusion and blocking conflicts.

6.3.3 Remote procedure invocation pattern

How to integrate multiple applications so that they work together and can exchange information? The applications are built independently, using different languages and platforms. The applications need more than just sharing data, as in the File transfer and Shared database patterns.

The *Remote procedure invocation pattern* enables the applications to share functionality, not just data. In the Remote procedure invocation (RPI) pattern, each application is a separate component with encapsulated data, which provides an interface to allow other applications to interact with the running application. Figure 6.4 shows a UML deployment diagram where two components have a remote procedure invocation.

A number of technologies, such as Unix RPC (remote procedure call), JSON-RPC, CORBA or Java RMI (remote method invocation) implement the Remote procedure invocation pattern. Standards such as SOAP also allow RPI-style Web services.

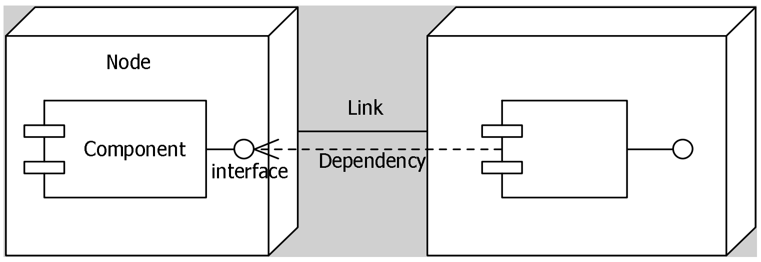

Figure 6.4: Remote procedure invocation

RPI is more than just sharing data. The Shared database pattern provides an unencapsulated data structure. In order to apply object-oriented techniques like information hiding, we want to hide the raw data behind access methods, but this behavior needs to be copied at every location. If we want to centralize the functionality of the Data mapper pattern, we need a way to call its methods remotely.

Advantages

RPI makes it easier to deal with semantic dissonance: a component may provide separate interfaces for different kinds of clients. Also, it is easy to program, because remote procedure or method invocation mimics local procedure or method calls.

Disadvantages

The fact that the Remote procedure invocation pattern mimics local procedure invocation is a disadvantage at the same time: it hides the big differences between those two with respect to reliability and performance.
Also, the Remote procedure invocation pattern involves tight coupling. This is due to synchronization, because one component waits while the other completes the procedure invocation, and due to interface dependency, which makes it difficult to change and develop independently.
The remote calls tend to tie the different system into a growing knot. In particular, sequencing - doing certain things in a particular order can make it difficult to change systems independently. People often design the integration the way they would design a single application, unaware that the rules of the engagement change dramatically [24].

6.3.4 Messaging pattern

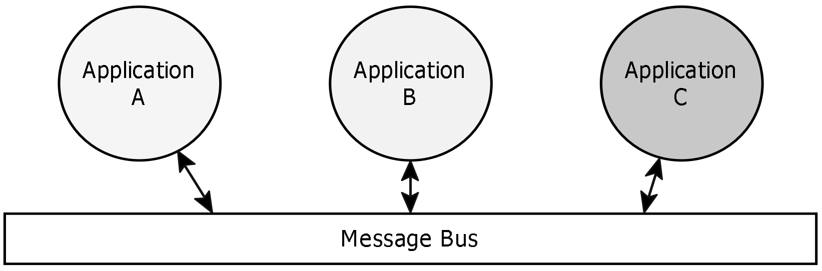

Figure 6.5: Messaging

Let an enterprise have multiple applications that are built independently, in different languages and platforms. The enterprise needs information to be shared rapidly and consistently. In OO, method calls are sometimes loosely called messages. That is not what we mean here. We mean real messages, like e-mails. The big difference is that with a method call, you wait for the answer before proceeding. With a real message, you send it and move on. It is like the difference between a phone conversation and voicemail.
In this case the asynchronous messaging is fundamentally a pragmatic solution of the integration problem. The asynchronous paradigm *send and forget* does not require to both systems to be up and ready at the same time. One application's failure does not bring down all of the other applications.

Advantages

There are some pragmatic advantages of the Messaging pattern.

– Messaging does not require both systems to be up and ready at the same time.
– This pattern forces developers to develop components with high cohesion. Since remote messages are costly, the developers will try to minimize messages and concentrate strongly related jobs in one component. In the context of messaging high cohesion means that lots of work is done locally, and only selective work is done remotely. Remote calls do not masquerade as local calls.
– Messages can be transformed in transit without sender or receiver having to pay attention to this.
– Messages can be broadcast to many receivers.
– Application development can proceed separately from integration decisions.

Moreover, the Messaging pattern allows applications to collaborate behaviorally, i.e. allows process to be launched at once if a certain event takes place. Remote procedure invocation also allows applications to collaborate behaviorally, but blocks the whole system while this is going on.
Messaging allows much of the decoupling, like the File transfer pattern, but with rapid updating, automatic retry until success, and automatic notification. Messaging is not prone to failure and deadlock.

Disadvantages

Messaging reduces but not entirely eliminates the time lag problems. These problems are not so serious as in the File transfer pattern, but still exist. The asynchronous solution means that systems are not being updated at quite the same time (e.g. the "Midnight problem": time checked at $23:59$, date checked at $00:01$).
Messaging itself does not provide a solution to the semantic dissonance problem, however the problem can be solved by sending messages to a data mapper.

Basic messaging concepts

Messaging involves the following basic concepts:

- A *channel* is a virtual pipe that connects a sender to a receiver. A newly installed messaging system typically does not contain any channels: you must determine how your applications need to communicate and then create the channels to facilitate this [24].
- A *message* is an atomic packet of data that can be transmitted on a channel. To transmit data, an application must break the data into one or more packets, wrap each packet as a message, and send the message on a channel. The opposite process is implemented at the other end. The messaging system will repeatedly try to deliver the message until it succeeds [24].
- *Pipes and filters*. In the simplest cases the messaging system delivers a message directly from the computer of senders to the computer or receivers. In more complex cases, some actions have to be performed on the message by the messaging system, e.g. validation, or transformation to a different format. The Pipe-filter pattern describes how multiple processing steps can be chained together using channels.
- *Routing*. In a large application a route of a message can be complex. A message can go through several channels and to manage the routing a message is sent not to the final destination but to a router. A *router* is an application component that takes the place of a filter in the Pipe-filter pattern. It determines how to navigate the channel topology and directs the message to its final receiver, or at least to the next router.

- *Transformation*. Various applications may not agree on the format for the same conceptual data. To reconcile this, the message must go through a filter, a message translator which converts the message from one format to another.
- *Endpoint*. In order to use messaging, most applications must contain a layer of code that knows how the application works and how the messaging system works. This layer of code is a set of message endpoints that enables the application to send and receive messages.

Message channel subpattern

A *channel* is a logical address in the messaging system. Logical addresses are characterized by an alphanumeric name, often within a hierarchical organization. One physical channel can contain many logical channels as long as it knows how to keep them separate. Applications use different message channels for different types of information.

Messaging systems do not come preconfigured with message channels. They must be configured by the system administrator to set up the channels that the applications expect. Consider, for example, a message channel in the *Java Messaging System* (JMS). The Sun J2EE SDK ships with a reference implementation of the J2EE services including JMS. The `j2eeadmin` tool can configure message channels:

```
j2eeadmin -addJmsDestination jms/myqueue queue
```

Once the channel has been created, it can be accessed from JMS client Java code:

```
Context jndiContext = new InitialContext();
Queue myQueue =
(Queue) jndiContext.lookup("jms/myqueue");
```

Publish-subscribe channel subpattern

How can the server broadcast an event to all interested receivers? The *Publish-subscribe channel pattern* expands upon the *Observer pattern* by adding the notion of an event channel for communicating event notifications.

An event can be packaged as a message. The event channel is a message channel. Each subscriber has to be notified of a particular event once and should not be notified repeatedly of the same event. The event cannot be considered consumed until all the subscribers have been notified, but once they have, the event should disappear from the channel. Concurrent subscribers should not compete but should be able to share the event message.

The Publish-subscribe channel pattern has one input channel that splits into multiple output channels. When an event is published into a channel, a copy of the message is delivered to each of the output channels. Each output channel has only one subscriber. This way each subscriber gets the message only once and consumed copies disappear from their channels.

The Publish-subscribe channel can be useful debugging tool. Monitoring al traffic on a channel without disturbing the existing message flow is helpful for debugging.

Subscriptions need to be restricted by security policies if we do not want to allow anyone to write a simple program to listen to the message traffic. See also the Event bus pattern (module on Architectural patterns).

Message subpattern

Data to be transmitted have their structure: records, objects, database rows, etc. The structured data are sent as a stream of bytes. The first process must marshal the data into byte form, the second process must unmarshal the data back into its original form.

A message consists of a header and a body. A *message header* contains information used by the messaging system. A header describes such data as data origin, destination, type etc. A *message body* contains bytes being transferred. A message body is ignored by the messaging system.

Message router subpattern

In the Pipe-filter pattern, filters are connected directly to each other with fixed pipes. A Mmssage channel may contain messages from different sources that may have to be treated differently based on the destination or type of the message or other criteria.

The *Message router* is like a filter, except that it

- connects to multiple output channels,
- does not modify message contents, only concerned with destination,
- decision criteria for the destination of a message are maintained in a single location.

If new message types are defined, new processing components are added, or routing rules changed, we need to change only the message router logic, while all other components remain unaffected.

The Message router pattern has the following variants:

- A *content-based router* decides the message destination only on properties of the message itself (e.g. type, or values of specific fields).
- A *context-based router* decides the message destination based on environment conditions (e.g. load balancing, failover functionality). *Failover functionality*: If a processing component fails, a context-based router can reroute messages to another processing component.
- A *stateless router* looks at only one message at a time.
- A *stateful router* takes the context of previous messages into account (e.g. to eliminate duplicate messages).

The notion of a message router is central to the concept of the *Message broker pattern*, implemented in for instance JBoss Messaging or Apache ActiveMQ (see the Broker pattern in the Architectural patterns module). Such EAI tools accept incoming messages, validate them, transform them, and route them to the correct destination. The architecture ensures that participating applications need not be aware of one another, because the Message broker pattern brokers between the applications. This is a key function in enterprise integration because most applications to be connected are packaged or legacy applications and cannot be changed.

Message endpoint subpattern

An application and the messaging system are two separate sets of software. A messaging system is a type of server, capable of taking requests and responding to them, accepting and delivering messages. An application that uses messaging is a client of the messaging server.

The messaging server has an API that the application can use to interact with the server. The API is not application-specific but is domain specific, where the domain is messaging. The application is probably not aware of this API, and should not be vulnerable to changes in it. The application should contain a set of code that connects and unites the messaging domain with the application to allow the application to perform messaging.

A *message endpoint* is a set of code specific to both the messaging API and the application, that shields the application from the messaging system. The Message endpoint pattern encapsulates the messaging system from the rest of the application and customizes the general messaging API for a specific application.

A message endpoint can be used to send messages or receive them, but one individual message endpoint object does not do both. An endpoint is channel specific, so a single application would use multiple endpoints to interface with multiple channels. An application might even use multiple endpoints at a single channel, usually to support multiple concurrent threads.

A message endpoint is designed as a specialized messaging gateway (see module on Enterprise Application Architecture).

In JMS the two main endpoint types are `MessageProducer`, for sending messages, and `MessageConsumer`, for receiving messages. A message endpoint uses an instance of one of these types to either send messages to or receive messages from a particular channel.

Polling consumer subpattern

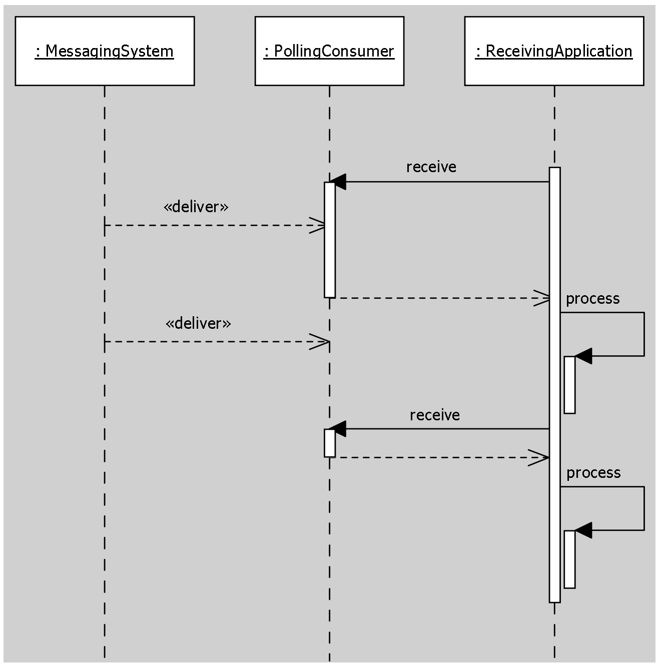

Figure 6.6: Polling consumer

How does a message endpoint know when a new message is available? How can an application consume a message when the application is ready?

A simple solution is to repeatedly check the channel to see if a message is available. When a message is available, it consumes the message and then goes back to checking for the next one (Figure 6.6). This process is called *polling*. The advantage of this approach is that the consumer can request the next message when it is ready for another message. So it consumes messages at the rate it wants to rather than at the rate they arrive in the channel.

In JMS, for example, a message consumer uses `MessageConsumer.receive` to consume a message synchronously. MessageConsumer has three different receive methods:

- `receive()`: This method blocks until a message is available and then returns it;
- `receive(long)`: This method waits for the next message that arrives within the specified timeout interval.
- `receiveNoWait()`: This method checks once for a message and returns it or null;

Event-driven consumer subpattern

If the channel is empty, a polling consumer will spend processor time looking for messages that are not there. The alternative to this solution is to let the channel tell the client when a message is available. It is also known as an *asynchronous receiver*, because the receiver does not have a running thread until a callback thread delivers a message. An event triggers the receiver into action.

An *event-driven consumer* is an object that is invoked by the messaging system when a message arrives on the consumer's channel (Figure 6.7). The messaging system also invokes an application specific callback. An Event-Driven Consumer has an application specific implementation that confirms to a known API defined by the messaging system.

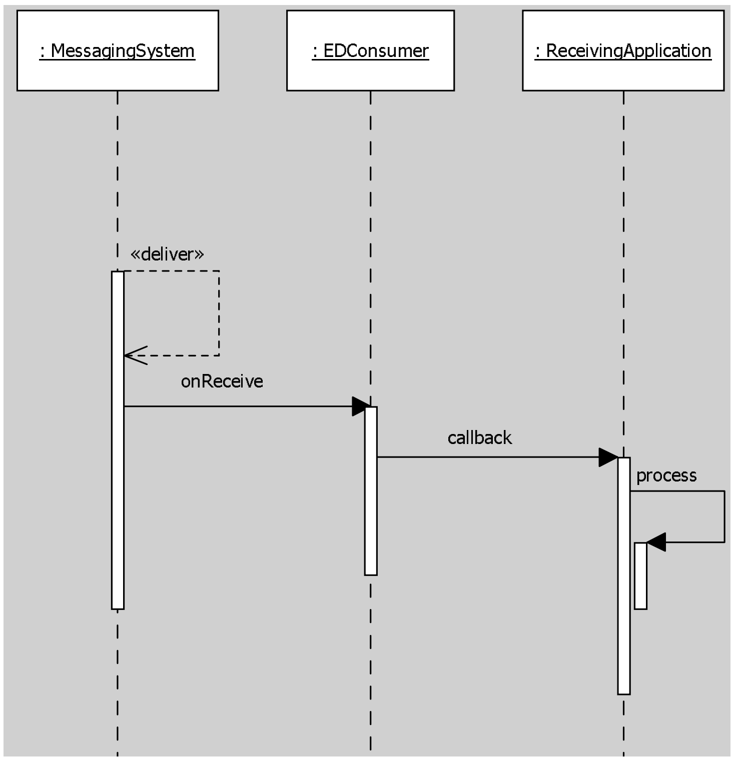

Figure 6.7: Event-driven consumer

In JMS, an event-driven consumer is a class that implements the `MessageListener` interface with method `onMessage()`.

6.3.5 HornetQ, open source message oriented middleware

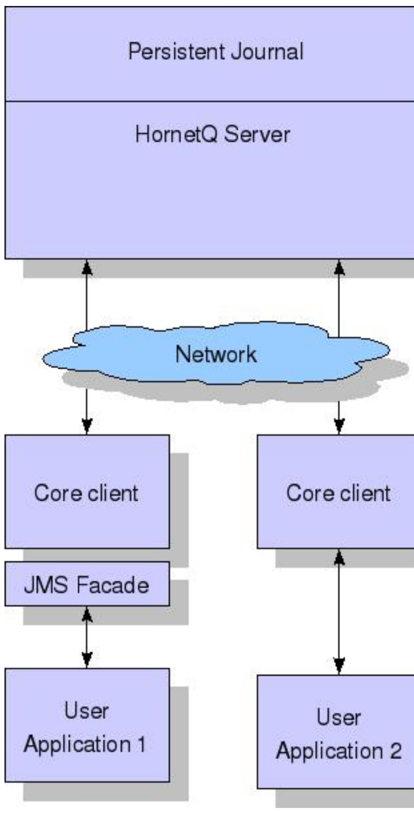

Figure 6.8: HornetQ Message Oriented Middleware

The HornetQ system (Figure 6.8) from JBoss supports many subpatterns from the Messaging pattern: it supports both publish-subscribe as point-to-point messaging, both blocking and non-blocking sends, scheduled messages, message priorities, and both persistent and non-persistent messages. It provides guaranteed delivery, i.e. ensures that messages arrive once and only once where required, and supports ACID ((atomicity, consistency, isolation, durability) semantics, and has a customizable security framework based on JAAS. Messages can be ordered, which means that they will be delivered in exactly the same order as they have been placed in the queue. Clients and servers can be configured to communicate using unencrypted TCP sockets, encrypted TCP sockets using the Secure Sockets Layer (SSL), over HTTP requests and responses also behind a firewall), or within the same Java virtual machine.

The system provides a management API that can be accessed using JMX.

6.4 Conclusion

“When applying an integration pattern, an application developer should avoid getting bogged down by standards and stay focused instead on the particular use cases at hand. ... As standards mature, architects are wise to consider how their use might extend enterprise integration solutions to broader, less risky, less costly and more powerful level of sophistication” [24].

DISCUSSION QUESTIONS

1. We stated that common standards of presentation do not imply common semantics. Can you think of examples? Can you think of remedies?
2. What would be good strategies to minimize coupling between different applications?
3. Many e-mail systems malfunction whenever a message body happens to start with the word "From". What is wrong here?

Block III

Components and contracts

Module 7

Component software

INTRODUCTION

The traditional way of developing software is to develop entirely from scratch, using only the programming tools and libraries. The software developed this way is called custom-made software. On the one hand, custom-made software has a lot of advantages. It is optimally adapted to the user's business model, it uses optimal solutions in the local areas of expertise and takes advantage of domain knowledge. On the other hand, custom-made software is very expensive. Custom-made software will have sub-optimal solutions in all but the local areas of expertise, and chasing state-of-the-art solutions is a burden. It involves a risk of failure and of late delivery, and interoperability with business partners or customers is not guaranteed.

The opposite of custom-made software is standard software, produced for a mass market. Standard software can be parameterized to provide a solution that is close enough to the business needs, and as such covers the time-to-market risk. On the other hand, standard software may necessitate business process reorganization, does not provide a competitive edge, and cannot be quickly adapted. For example, some ERP (enterprise resource planning) software fixates the hierarchic relations between departments and therefore puts an end to the autonomy of departments.

Component software represents a solution in between. Component software is easier to customize than standard software.

LEARNING OBJECTIVES

After having studied this module you should be able to:

- to explain the goal of component software,
- to describe what a component is,
- to describe standards of component connecting,
- to make a comparison between web services and components.

7.1 Custom-made, standard and component software

DEFINITION 7.1 *Software components* are binary units of independent production, acquisition and deployment that interact to form a functioning system.

An often used word is *COTS*, originally meaning: Commercial Off-The-Shelf Software: software that can be reused by other software products to become part of their product. COTS is now used to denote both commercially available and open-source software components [22]. A COTS product has the following characteristics:

- it is not produced here, i.e. within these projects nor exclusively for them;
- it may be closed-source as well as open source, being open source software usually treated as if it were closed;
- it is not a commodity, i.e. not shipped with the operating system, nor provided with the development environment, nor generally included in any pre-existing platform;
- it is integrated into the final delivered system, i.e. it is not a development tool;
- it is not controllable, in terms of provided features and their evolution.

Component assembly allows for individual investment/performance trade off choices, and components may be combined with custom-made components. The trade-off in

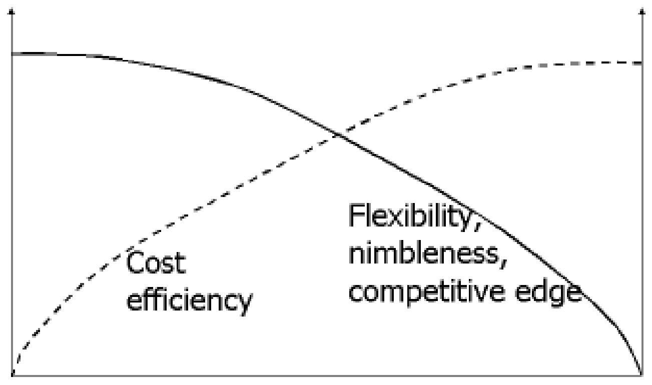

Figure 7.1: Custom-made or bought component

Figure 7.1 can be read as follows: when there are no demands for flexibility, changeability, being at the competitive edge and so on, a COTS component is very cheap compared to in-house development of such a component. However, when the demand for changes is high, it is cheaper to develop and maintain a component in-house than to choose a COTS component and look for a replacement each time a change is needed.

7.2 The nature of components

Software components may be compared with integrated hardware circuits, blocks of stereo equipment or nuts and bolts, but software is different: it does not deliver a product but a blueprint. A programmer provides classes that are instantiated to objects at runtime.

A component is a *unit of deployment*; it is an isolatable part of a system. It is the smallest part of a system about which one can decide whether to use it or not. A component may look like a collection of classes from the outside (it only rarely looks like a single class); it may be programmed in a non-object-oriented language.

Examples of components are:

- Every application running on an operating system is a coarse-grained component. Interaction between these components is carried out by common file formats (a .gif file for instance, is readable and editable in a number of applications) or by pipe/filter composition (in Unix, filter applications like sort or uniq expect a stream of characters as input and have a stream of characters as output, and can be composed using pipes, in a shell script or on the command-line).
- In a microkernel architecture OS, some functionality is delegated to application-level services. These services may be seen as components. Examples of such an architecture are Symbian or L4.
- In plug-in architectures, plug-ins are components. Examples are the Mozilla browser or the Eclipse framework. In all cases, it is possible to extend the functionality by adding plug-ins.
- JavaBeans are also an example of components.

7.2.1 Modularity

Components provide *late integration*: they are integrated after acquisition and deployment. This also means that integration testing is no longer feasible: the number of combinations in which a component can be used is infinite, so testing all those combinations in advance is impossible.

What is possible is modular checking: analysis of component properties based only on the interfaces explicitly required. Also, if possible, one can inspect the code of a component and check whether all assumptions about its environment are explicitly stated, and check whether the number of assumptions is as small as possible. This requires defensive programming: to program without assumptions, or at least with as small a number of assumptions as possible.

7.3 Components and objects

7.3.1 Characteristics of components

Components have the following characteristics:

- A component is a unit of independent deployment: it is well separated from its environment and from other components, and it will never be deployed partially.
- A component is a unit of third-party composition: it has to interact through well-defined interfaces.
- A component has no persistent state: a database server is component, a database is an object. The reason behind this is that we do not want to discern different instances of a component: a component is comparable to a class, not to an object.

7.3.2 Characteristics of objects

Objects have the following characteristics:

- An object is a unit of instantiation: it cannot be partially instantiated.
- An object has a unique identity that separates it from other objects. This identity does not change with changes of its state during its lifetime. For instance, when a person is represented as an object, the object may get a new address and a new name, but it will still represent the same person, because of its unique identity. An object is created from a plan (called a class) or from a prototype object.

– An object encapsulates a state. Its initial state is provided by a constructor or a factory. The state changes during the objects lifetime.

7.3.3 Components and classes

A component usually consists of a collection of *classes* (like a Java package). However, it is not always a collection of classes inside: it may be implemented in a functional language or assembly language. A class is necessarily part of only one component.

7.4 Reuse

Software units can be reused in two ways: blackbox and whitebox reuse.

DEFINITION 7.2 *Blackbox reuse* is reuse of a software unit knowing only the interface and its specification.

DEFINITION 7.3 *Whitebox reuse* is reuse of a software unit knowing its source code.

An example of black-box reuse is using an API (application programming interface). An example of white box reuse is using class libraries or frameworks. Reuse of components is blackbox reuse: open software components for instance may be used without knowledge of the source code, even though the code is available.

7.5 UML and components

7.5.1 Component diagrams

DEFINITION 7.4 A *component* in UML stands for a replaceable part of a system that conforms to and provides the realization of a set of interfaces.

Interfaces

Components can have two kinds of interfaces:

– *Provided interfaces:* those that the component provides as a service to other components, and
– *Required interfaces:* those that the component conforms to when requesting services from other components.

The model element for components has changed in UML 2.0 to accommodate required interfaces, hierarchical decomposition and ports.

Good components define crisp abstractions with well-defined interfaces, making it possible to easily replace older components with newer, compatible ones. Interfaces bridge your logical and design models. For example, you may specify an interface for a class in a logical model, and that same interface will carry over to some design component that realizes it. Interfaces allow you to build the implementation of a component using smaller components by wiring ports on the components together.

Figure 7.2 presents an example *component model*, using the UML 2 notation. In UML before version 2.0, the model element for a component looked like the small icon in the upper right hand side, and required interfaces (now represented by "sockets") were not shown. The model element was changed because a component is now regarded as a special case of a structured class: there really is not much of a semantic distinction between classes and components. It is often useful, however, to use the component

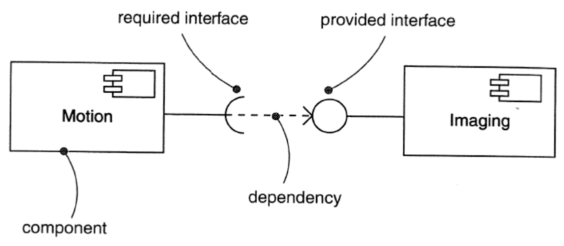

Figure 7.2: UML component diagram

notation for replaceable units, especially if these do not map directly to a single class in the implementation.

UML 2.0 components are modeled as simple rectangles. UML 2.0 uses the component presentation of UML 1.x as a visual stereotype within the rectangle to indicate that the rectangle represents a component (but a textual stereotype is also acceptable). A component diagram models dependencies, either between components or between components and interfaces. The lollipop symbol indicates a provided (implemented) interface and the socket symbol indicates a required interface.

Ports

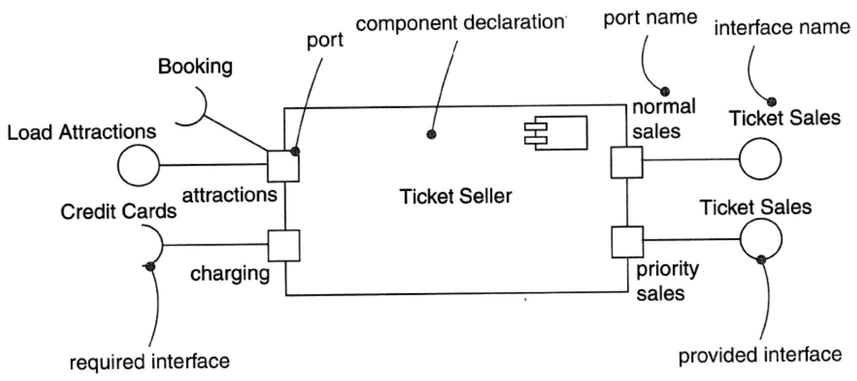

Figure 7.3: Ports

In an *encapsulated component*, all of the interactions into and out of the component pass through *ports*. A port can be seen as an explicit window into the component. A port has a uniquely identifying name. The externally visible behavior of the component is the sum of its ports.

Ports permit the interfaces of a component to be divided into discrete packages that can be used independently. The internal parts of a component may interact through a specific external port, so that each part can be independent of the requirements of the other part.

Figure 7.3 shows that the provided interface *Load Attractions* and the required interface *Booking* are combined together into the port *attractions*.

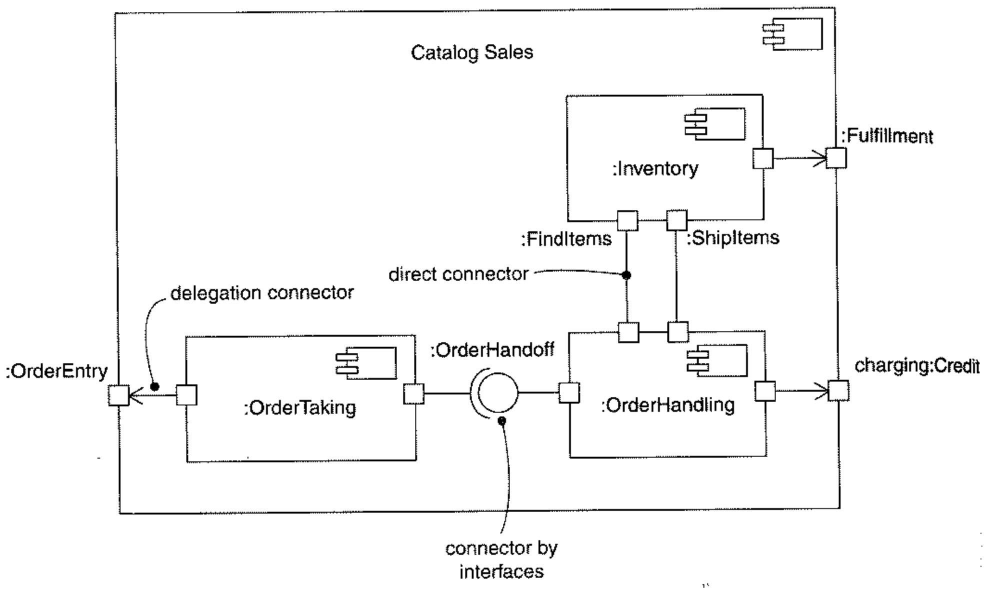

Figure 7.4: Connectors

Connectors

It is desirable to build large components using smaller components as building blocks. The internal structure of a component consists of the parts that compose the implementation of the component together with the connections among them. A *connector* is a wire between two ports. It represents either a link (association) or a transient link (supplied by a procedure parameter).

If two components are explicitly wired together, either directly or through ports, we draw a line between them or their ports as it is shown in Figure 7.4 between the components *Inventory* and *OrderHandling*. On the other hand, if two components are connected because they have compatible interfaces, we use a ball-and-socket notation to show that the components have no inherent relationship, although they are connected inside this component. Figure 7.4 shows such a connector: *OrderHandOff* between components *OrderTaking* and *OrderHandling*.

You can substitute one component by another component if they both satisfy the same interface.

7.5.2 UML artifact diagrams

Artifacts are things that live in the world of bits, e.g. executables, libraries, tables, files, or documents. *Artifact diagrams* are used:

- to model source code, for instance C++ header- and source files, and their compilation dependencies,
- to model executable releases, for instance DLL's (Dynamic Link Libraries) manifesting components,
- to model physical databases, for instance a set of tables,
- to model adaptable systems, for instance databases that get replicated across several servers.

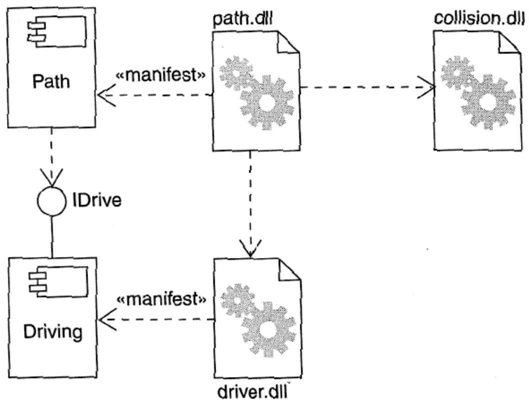

Figure 7.5: Artifacts

Figure 7.5 shows an example of an artifact diagram, in UML 2.0. An artifact *path.dll* and component *Path* have an association *manifest*. An artifact is usually depicted as a document or a box with one of the stereotypes *Executable*, *Library*, *File* or *Document*.

7.5.3 UML deployment diagrams

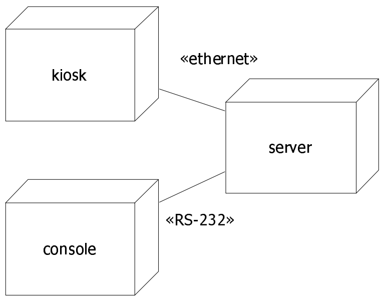

Figure 7.6: UML deployment diagram

A UML 2.0 *deployment diagram* (Figure 7.6) depicts a static view on the run-time configuration of processing nodes and the components that run on those nodes. A *node* is a physical object that exists during runtime and represents a computational unit (e.g. a computer, a printer or a router). A node is represented as a 3D rectangle. A *connection* is a physical connection between nodes (e.g. an Ethernet connection).

The nodes of deployment diagrams depict physical objects, as do artifacts, but artifacts are things that participate in the execution of a system; nodes are things that execute artifacts. Artifacts represent the physical packaging of other logical elements; nodes represent the physical deployment of artifacts. The relationship between a node and the artifacts it deploys can be shown explicitly by nesting (but most of the time we rely on text-based information).

7.6 Concepts associated with components

7.6.1 Components

DEFINITION 7.5 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 [54].

Components should specify their needs, called *context dependencies*, referring to the context of composition and deployment. In the real world, the producers of components emphasize their provided interfaces (as opposed to their required interfaces). Several component worlds exist, that partially conflict, such as CORBA, Universal Network Objects from OpenOffice, D-Bus (Gnome), or J2EE. These "worlds" consist of conventions about how distributed components may communicate.

7.6.2 Component weight

Reducing context dependencies of a component means that more required software comes bundled into the component, which means that the *weight* of the component is increased. Ultimately, this defeats the purpose of using components: increasing the weight of a component eventually leads to providing an entire application.

On the other hand maximizing reuse by minimizing the component's weight means that everything but the prime functionality is left out. This leads to an explosion of context dependencies.

7.6.3 Component interface

DEFINITION 7.6 A *component interface* is the set of named operations provided by this component for invocation by clients.

Every operation in the interface must be specified. The specification provides the communication between the implementor and the clients. A specification consists at least of the name, signature and return type. But see also below: contracts about behavior must be added for safety.

Direct and indirect interfaces

A *direct interface* couples a name of a procedure to an implementation, just as a function or procedure is called by a single name. A component may provide a set of procedures that are used as procedures of a procedure library would be used: the name of a procedure points to one implementation.

An *indirect interface* may point to different implementations, just as a method of a class may point to different implementations, depending on the class of the object that implements the interface. (Direct interfaces can be viewed that way too: a direct interface can then be viewed as a static method.)

Indirect interfaces introduce indirection via method dispatch. Such a late binding depends on the quality of interface specification.

Versions

With direct interfaces, version numbers may be checked at bind time. With indirect interfaces, version checking is not sound, because object references may cross component boundaries, so it is not possible to check the version of an implementation at bind-time.

For example, suppose a class C, with two incompatible versions C_1 and C_2. Component A needs C_1, while component B needs C_2. It is possible that A sends an object to B, with type C. When the object was created, A has checked its version, which was C_1, so A believes everything is in order. Component B now only sees the static type of the object, which is C. Unfortunately, B is no longer able to check the object's version, because it cannot determine the dynamic type.

Interfaces as contracts

An interface specification may be viewed as a *contract* (more on contracts in a separate module) between a client and a provider. A contract binds both parties; it also states what the client has to do to use the interface correctly. Hence the interface is not required to be foolproof.

The functionality of an operation is specified by pre- and postconditions. These can be separated in *safety* (nothing bad can happen) and *progress* (something good will happen). In a contract, nonfunctional requirements are also important:

- availability,
- mean time between failures,
- mean time to repair,
- throughput: average amount of work done per unit of time,
- latency: the maximum time the client has to wait before a request is processed (hospital polyclinics have high throughput, but high latency as well),
- data safety,
- space and time requirements.

Undocumented features, not in the contract but observed, e.g. by debugging, are not to be used because they cannot be relied on in future versions.

7.6.4 Independent extensibility

The principal function of component orientation is to support *independent extensibility*, which means that independently developed extensions can be combined (think of browsers or applications as Eclipse for which plug-ins may be developed independently).

A problem with independent extensibility is that the OS must ensure that security policies are respected: proper authorization must be used, there should be a hardware protection mechanism, and there must be a mechanism to use cached content. Also, the runtime overhead may be up to a factor 100.

7.6.5 Safety by construction

There are techniques to enhance the chance that an independently developed extension behaves properly with respect to security:

- Java uses static type-checking. It provides automatic memory management, and runtime checks on array boundaries. Therefore, programs like Together or Eclipse, written in Java, can guarantee that memory errors cannot occur.
- Another possibility is to define an intermediate language to which a wide variety of programming languages can be compiled. This intermediate language can have strong safety properties, established at compile-time, and limits the used languages to their safe subsets.

Using these techniques for *safety by construction* guarantees safety properties that can be relied on without further checking at runtime.

Module safety

Unfortunately, type safety and elimination of memory errors are not enough: with these in place, it is still possible for a program to call arbitrary services present in or loadable into the system. A downloaded component might, without any further safety measures, take control of the computer, or make use of communication ports.

Module safety means that a component has to specify explicitly which services (from the system or other components) it needs to access. It works in the same way as access control in file systems, but on a module or class basis. In Java, for example, the `import` statement guarantees that the class only retrieves references to other objects which it has imported through the import statement. Accessing objects of classes that are not mentioned in an import statement is impossible.

7.7 Connecting components

Binary calling conventions at the procedural level are usually not supported across process boundaries. Two running Java programs, for instance Together and Eclipse, cannot simply call a method of an object in the other program.

A wide variety of mechanisms for *interprocess communication* exists:

– communication using files that can be accessed by two or more processes,
– communication through pipes, as in the pipe and filter pattern, seen in Unix for example,
– communication through sockets, which is possible between two processes on different machines,
– shared memory as a means for communication.

In general, these solutions are not portable across platforms (in some cases, asynchronous communication through files is an exception). An advantage is that these solutions can easily be scaled over a network. They operate on the level of bits and bytes, and thus it is hard to implement complex interactions on top of them.

7.7.1 Remote procedure calls

Remote procedure calls were introduced 1983 [7]. The idea is to use procedure stubs at both ends. The caller of a remote procedure call simply uses local calling conventions, and superficially seems to call a local callee. In reality it calls a local *stub* that *marshals* (serializes) the parameters of the procedure and sends them to the remote end. At that end another stub receives the parameters and *unmarshals* (deserializes) them, and calls the true callee using local conventions (that may differ from those used by the initial caller).

RPC's are transparent to users. Stubs are automatically generated and neither client nor providers need to be aware of non-local calling. This is an advantage because it simplifies the programming model by mapping all levels of communication (in-process, interprocess, intermachine) onto a single abstraction. On the other hand, the transparency also hides cost differences between local, interprocess and intermachine calls. Intermachine calls are approximately a thousand times slower than local calls.

Support for remote procedure calls is offered in procedure libraries. These libraries must be distributed in source form, and are compiled together with the application using the libraries.

7.7.2 Interface Definition Languages (IDL)

Interface definition languages have been introduced to automate the generation of stubs. For each remotely callable procedure, an IDL specifies the number, passing modes, and types of parameters and of possible return values. To ensure that communication across machine boundaries works, an IDL fixes the range and binary coding conventions of basic types. An IDL may for instance marshal integers to big-endian 32-bit two's complement encoded integers[1]. An IDL also generates universally unique identifiers.

Examples of IDLs are the DCE IDL (Distributed Computing Environment, a standard of the Open Software Foundation), the OMG IDL (Object Management Group, international standards organization, founders of CORBA) and the IDL for the Universal Network Objects for OpenOffice.

7.8 Future and current approaches

7.8.1 Scala

Scala is a research project [36] with the goal to integrate full support for components into a programming language. It is interoperable with Java without glue code. Scala offers uniform abstraction concepts for both types and values, it unifies functional and object-oriented programming, and it has built-in support for pattern matching and XML.

Scala is designed to be scalable: the same concepts describe small as well as large parts. Everything is an object: operations, functions, values, components. To be reusable, a component should publish both its provided and its required interfaces. The latter has so far been missing in programming languages.

In Scala, components are classes and *traits*. A trait is like an interface in Java, with the possibility to have partially implemented methods: a trait is an abstract class without data members. All objects, including functions, are first-class values, which means that they can be passed around as a parameter, for instance. Scala supports recursive references, which means that for instance a function can use itself inside its implementation, and supports inheritance with overriding. Component composition is done by mixin-based class composition, which means that a class may inherit from one class. Components may use partial implementations of traits, similar to the use of interfaces in Java, with the difference that a class may also use implementations of a trait. No explicit "wiring" is needed. Provided services are modeled as concrete members, required services are modeled as deferred (abstract) members.

7.8.2 XML

XML (extensible markup language) was introduced in 1998 [9]. Originally, XML was designed for large-scale electronic publishing, but today, it plays an increasingly important role in the exchange of a wide variety of data on the web and elsewhere.

XML is useful for representing any structured and semistructured data. Where relational databases prefer tabular data, XML prefers tree-shaped data. It is more flexible than relational databases.

XML is a text-based, generic approach to describing data for any purpose. Names are tagged unambiguously by *namespaces*. In Figure 7.7, there is a namespace denoted by `n`, with value `http://myorganisation.com/customer`.

XML is now commonly used for messages, documents, and configuration data. Most browsers directly support displaying XML documents. XML has become the "lingua franca" among applications of independent origin. It allows for independent extension

[1]Two's-complement is a system in which negative numbers are represented by the two's complement of the absolute value. The two's complement of a 32-bit binary number is the value obtained by subtracting the number from the 32 power of two

```
<?xml version = "1.0" ?>
<n:customer xmlns:n = "http://myorganization.com/customer">
  <n:CustomerID s = "313">
  <n:NameAdd>
    <n:Name s = "Donald Duck"/>
    <n:Address s = "Cornelis Prulplantsoen 1"/>
  </n:NameAdd>
</n:customer>
```

Figure 7.7: XML document

that does not interfere with documents that were created before such extensions were introduced.

The success of XML as a standard format for durable data representation is partly because of its properties, and partly because of proper timing: standardization was felt to be needed. XML is rooted in the older SGML (Standard Generalized Markup Language, 1986) but stands out because of its full support for extensibility.

The structure of XML

XML elements are opened and closed by tags, in the same way as is done in HTML. An XML element may contain nested elements or unstructured text. The order of elements is relevant. *XML attributes* appear within opening tags, again, in the same way that HTML elements may have attributes. Attributes are recognized by their names, and order is irrelevant

```
<?xml version = "1.0" ?>
<xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema">
  <xmlns:n = "http://myorganization.com/customer">
  <xsd:element name = "n:Customer" type = "n:CustomerType"/>
  <xsd:complexType name = "n:CustomerType">
    <xsd:sequence>
       <xsd:element name = "n:CustomerID" type = "xsd:string"/>
       <xsd:element name = "n:NameAdd" type = "n:NameAddType"/>
       <xsd:element name = "n:Remarks" minOccurs = "0"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name = "n:NameAddType">
    <xsd:sequence>
      <xsd:element name = "n:Name" type = "xsd:string"/>
      <xsd:element name = "n:Address" type = "xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Figure 7.8: XML Schema

Rules constrain well-formed XML documents. These rules were originally expressed in *DTDs* (Document Type Descriptors). DTDs are syntactically cryptic, and are not extensible. To overcome these disadvantages of DTDs, *XML Schema* has been introduced in 2001. XML Schema [17] is itself extensible, and schema's expressed in XML Schema are also extensible. An XML Schema is much richer and more expressive than a DTD.

XML Schema supports instance-level annotations and explicit attribution of elements with types. A schema document is itself an XML document, and therefore it can be subject to a *metaschema*. The XML Schema standard defines the metaschema for all schema documents, using XML Schema itself. Figure 7.8 shows an XML schema.

An *XML namespace* groups a set of names. That makes it possible to have unique names (in the same way that names for Java classes are unique). For convenience, a namespace is often referred to by a local alias: in Figure 7.8, `n` is the local alias for `http://myorganization.com/customer` and `xsd` is the local alias for `http://www.w3.org/2001/XMLSchema`.

When a URL is used as a namespace, this does not mean that the URL is used to gather data: URL's are only used as a means to generate unique names. Often, the URL points to an existing document, but even if it does not, the URL still does provide a unique name.

Simple Object Access Protocol (SOAP)

The *Simple Object Access Protocol* (SOAP) is an XML-based standard for invocations to remote objects, and was introduced in 2000. SOAP uses HTTP. It offers means to describe the addressee of an invocation and to encode programming data types.

SOAP does not guarantee that URL's yield the same object every time. This feature was intended to eliminate fragile state dependence: the situation that the caller assumes that the state of the object it calls remains the same between two subsequent calls. Such assumptions are only appropriate for synchronous communication between tightly coupled processes, and not for asynchronous communication over the internet.

The latency is about 10 times the latency of traditional RPC, due to the encoding in XML which is less compact than in traditional RPC. However, this increase in latency is negligible on wide-area or internet connections.

```
<SOAP-ENV:Envelope>
  <SOAP-ENV:Body>
    <xmlns:n = "http://myorganization.com/customer"/>
      <n:getNameAdd>
        <n:CustomerID s = "313"/>
      </n:getNameAdd>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

<SOAP-ENV:Envelope>
  <SOAP-ENV: Body>
    <xmlns:n = "http://myorganization.com/customer"/>
    <n:getNameAddResponse>
      <n:Name s = "Donald Duck">
      <n:Address = "Cornelis Prulplantsoen 1">
    </n:getNameAddResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 7.9: SOAP

A SOAP invocation corresponds to passing a message. The message is an XML-document made up of a mandatory SOAP envelope defining the message, an optional SOAP header that may present the processing instruction, and a mandatory SOAP body with call or response information. Figure 7.9 shows an example of a SOAP message and the corresponding SOAP response.

7.8.3 Web services

Web services offer computational services to other systems. Those other systems may be software, or websites targeting people. The computational services that are offered include databases.

A web service combines a deployed component and a service provider. The pricing structure may be very flexible. Web services are more self-contained than typical components or applications: they can not make any assumptions about the environment in which they will be used.

Web Services Description Language (WSDL)

```
<?xml version = "1.0" ?>
<definitions name = "StockQuote"
  targetNameSpace = "http://myorganization.com/stockquote.wsdl"
  xmlns:tns = "http://myorganization.com/stockquote.wsdl"
  xmlns:xsd1 = "http://myorganization.com/stockquote.xsd"
  xmlns:soap = "http://schemas.xmlsoap.org/wsdl/soap"
  xmlns = "http://schemas.xmlsoap.org/wsdl">
<types>
  <schema
    targetNameSpace = "http://myorganization.com/stockquote.xsd"
    xmlns = "http://www.w3.org/2001/XMLSchema">
    <element name = "PriceRequest">
      <complexType>
        <all><element name = "tickerSymbol" type = "string"/></all>
      </complexType>
    </element>
  </schema>
</types>
<message name = "GetPrice">
  <part name = "body" element = "‘xsd1:PriceRequest"’>
</message>...
```

Figure 7.10: WSDL

The *Web Service Description Language* (*WSDL*) from 2001 is used to describe a web service, much like an Interface Definition Language. It is built on XML. Figure 7.10 shows an example of a WSDL document that describes a service that accepts requests for stock quotes.

Universal Description, Discovery and Integration (UDDI)

The *Universal Description, Discovery and Integration service* (*UDDI*) from 2001 is a directory service to discover web-services. It is itself a web service, approachable through SOAP. An inquiry to find a business registered under a certain name is shown in Figure 7.11. It does not directly use WSDL, but a mapping between the data models exists.

```
<?xml version = "1.0" encoding = "UTF-8" ?>
<Envelope
  xmlns = "http://schemas.xmlsoap.org/soap/envelope/">
  <Body>
    <find-business
      generic = "1.0" xmlns = "urn:uddi-org:api">
      <name> MetaBit </name>
    </find-business>
  </Body>
</Envelope>
```

Figure 7.11: UDDI

DISCUSSION QUESTIONS

1. The goal of the Scala project is to bring component definition and composition into the programming language, and eliminate the need for 'glue' and 'wiring'. The goal of web services is to provide a mechanism for description, discovery and use of components that is totally independent from the implementation technique. Are these goals as incompatible as they seem? Discuss their respective merits.
2. Applications themselves can be viewed as components existing within the OS framework. How do they handle the required interface problem?
3. The relationship between software architecture and software components has not been addressed explicitly in this module. How would you describe the relation?

Module 8

Design by contract

INTRODUCTION

Because an architecture describes a system as a configuration of communicating components, it is important to be able to specify the interfaces of these components in such a way that parts of the system may be developed separately and can be assembled in the end, as long as these parts are built according to the specifications. In this module, we will explore interface specifications in detail.

An interface specification is a contract between a client and the provider of the interface. It is a contract that binds both partners. A popular method to specify such a contract is to formulate pre- and postconditions. A precondition is a constraint that must be met before the specified function or method is executed; a postcondition is what is guaranteed if the function or method terminates normally. In a contract, not only functional requirements should be specified, but also nonfunctional requirements. In practice however, nonfunctional requirements are often neglected.

In this module, we will dive into pre- and postconditions, and discuss the problems of external references, self-interference and callbacks when guaranteeing a postcondition. We will show remedies for these problems as well.

It is important to understand under which conditions a contract applies in the case of polymorphism. We will introduce the concepts of substitutability, of types and subtypes, of covariance and contravariance to make that clear. We will also discuss different types of inheritance, and their consequences for contract specifications. And finally, we will show you the technique of assertions in the language Eiffel, and the technique of model-based specifications.

LEARNING OBJECTIVES
After having studied this module you should be able to:
- explain the concepts of pre- and postconditions,
- explain the concepts of external references, self-interference and callback, the problems concerning contracts, and the way to overcome these problems,
- explain the different kinds of polymorphism, and their relation to contract specification,
- explain the problem of the fragile base class,
- describe the methods of contract verification,
- describe the problems with contract changes.

8.1 Contracts

8.1.1 Pre- and postcondition

A specification of the form

```
pre U,
post V
```

means: if the specified method is called in a state where `U` is satisfied, it will terminate in a final state where `V` is satisfied. An example of a specification is:

```
pre not buffer.full(),
post buffer = old buffer + item
```

This contract states that if the specified method is called in a state where the buffer is not full, it will end in a state where the buffer holds the items that were held before the method was called, plus one more item.

The *precondition* is concerned with the value of the buffer before the method was called, and the *postcondition* with the value of the buffer after the method was called. However, as you see in this example, it is often easier to specify the postcondition while using the values that were valid in the precondition: in this case, the postcondition is expressed using the 'old' value of the buffer.

Several formal notations allow the specification of pre- and postconditions, such as *OCL* (Object Constraint Language) [34] or *Object-Z* [51]. OCL is the standard notation in UML to specify pre- and postconditions.

The example uses the notation that is used in Eiffel [16] (using `old` for the values as they are valid in the precondition). In OCL, you do not use `old buffer`, but you would use `buffer@pre`.

8.1.2 Frames and class invariants

DEFINITION 8.1 A *frame* explicitly lists what may be changed by a method call.

When one does not use a frame, to be precise, one should add `v = old v` to the postcondition for every variable `v` that is not changed by the method call.

DEFINITION 8.2 A *class invariant* is a constraint that is true for every object of the class at start and finish of method invocations. A class invariant is expressed as a relationship between values of the attributes of the class.

An example of a class invariant is

```
tax = country.getRate() * total
```

This means that before and after each call to a method of the class, the value of the attribute tax is equal to the value of the attribute total, multiplied by the value that the attribute country gives when its method `getRate` is called.

A class invariant is automatically added to both the pre- and postcondition for every operation, even for newly added operations in subclasses. A class invariant should even be valid at exceptional termination, when the normal postcondition is not reached.

8.2 Problems with contracts

It is not always easy to guarantee that a given function or procedure will behave as its contract says. Problems may occur with external references, self-interference, and callbacks. Here, we discuss each concept, show how it may cause problems, and show the remedies.

8.2.1 External references

Consider a class A that names an object of class B in its class invariant. How can we guarantee that operations on this object do not violate the invariant? This problem is known as the *indirect invariant effect*.

```
public class Man

    protected Woman wife;
      // inv wife.husb = this

    public void marry (Woman w)
      // frame wife, pre w ≠ null, post wife = w

         wife = w;
         w.marry(this);

...
```

In the example, the class invariant states that each *Man* is married to zero or one *Woman*, and that no *Woman* is married to more than one *Man* (there may be unmarried *Women* though): the *Woman* a *Man* is married to, should be married to that *Man*.
It is obvious that the statement *wife = w;* is sufficient to guarantee the postcondition of *marry*, but that statement can not guarantee the class invariant. In this case, to guarantee the class invariant, it seems necessary and sufficient to add the statement *w.marry(this);*. After all, the *Man* calls a method *marry* from his *wife*, with himself as a parameter, so you would think that the class invariant of *Man*, which states that he is the husband of his own wife, seems to be safe.
But the invariant will be broken when the wife marries someone else: then *w* may be in a state where *w.husb != this*. A *Man* has no complete control over *Woman*, so *Man* cannot guarantee that *wife* will not exchange him for another *Man*.

```
public class Woman

    protected Man husb;

    public void marry (Man m)
      // frame husb, pre m ≠ null, post husb = m

         husb = m;

...
```

The class *Wife* has a method *marry* which seems to reestablish *Man*'s class invariant: when he marries her, she marries him, so his class invariant *wife.husb = this* is true. But can we guarantee that it will be true all the time? Consider the following scenario:

```
Man m1, m2;
Woman w;
m1.marry(w);
w.marry(m2);
```

As a result of these statements, *Man m1* thinks he is still married to *Woman w*, but she is married to *Man m2*: the invariant *wife.husb = this* is violated for *Man m1* when *w.marry(m2)* is executed. We created *Woman* so that she would marry every man she is asked to marry, even if she is already married to another man: the method *marry* can be called by third parties, outside the two partners of the contract.

Remedy

A remedy for this problem is to only allow external references in class invariants if these class invariants are immune against operations on other classes. The remedy to the indirect invariant effect is similar to problems encountered with multithreading. In the case of threads, one may safely assume that it does not matter whether the state of a program is interfered with by other programs (or other threads), if the constituents of the proof of the program are not interfered with by other programs or threads [38]. The Owicky-Gries method says: consider two processes. One process affects *x* and *y*:

```
*[x:= 1]
*[y:= 1]
```

and the other process only affects *x*:

```
*[x:= 2]
```

In that case, you may not write *x=1* as a postcondition. However, you may write *y=1*.

In the *Man* and *Woman* example, this means that we have to add a precondition

```
husb.wife ≠ this
```

to the method *Woman.marry* (which prevents a *Woman* from marrying when she has already a husband), and a precondition

```
w.husb.wife ≠ w
```

to the method *Man.marry*. In the two statements above, first *m1.marry(w)* is called. The precondition of the method *marry* of *Man* says that the wife of the husband of *w* should not be *w*, which means that she is not already married. First the variable *wife* of *m1* gets the value of *w*, and then *w.marry(m1)* is called.

The precondition of that method says that the wife of the husband of *w* may not be *w*, which is not the case, because she is not yet married. After the execution of both methods, *m1.wife = w* and *w.husband = m1*: our man is married to his wife, and she is married to him.

Now we try to execute *w.marry(m2)*. The precondition says that the wife of the husband of *w* should not be *w* herself. But, as the husband of *w* is *m1*, and the wife of *m1* is *w*, the precondition is not fulfilled, which means we cannot execute *w.marry(m2)*.

8.2.2 Self-interference

```
public class DigitList

   int m;
   int[] a = new int[100];
     // inv (all i: 0 <= i <= m: a[i] < 10)

   public void add(int x)
     // frame m, a
     // pre x < 10
     // post m = old m + 1
     //   and a = old a ⊕ (m → x)
   {
        m++;
        print();
        a[m] = x;
   }
   ...
```

The invariant of the class *DigitList* says that each filled cell of the array *a* contains a value smaller than 10, and that the first *m+1* cells have a value. The precondition of the method *add* says that *x* is smaller than 10, and the postcondition says that 1 has been added to the old value of *m* and that the array *a* is the same as before the method was called, with one extra filled cell with the value of *x*.

It is obvious that the method *add* does not violate the class invariant. However, the method *print* is called at a moment in which the class invariant temporarily does not hold: the number of filled cells of the array *a* is 1 less than the value of *m+1* at that moment.

This is an example of *self-interference*: a state in which the class invariant temporarily does not hold is exposed through a method call of the object itself.

```
interface TextModel

   void write(int pos, char ch);
   void register(TextView x);
   ...

interface TextView

   TextModel text;
   void setCaret(int pos);
   void type(char ch);
   void insertNotification(int pos);
   ...

// implementation:
void type(char c)

   int p = caretPos();
   setCaret(p+1);
   text.write(p, ch);
```

Consider *TextModel* and *TextView* as part of a MVC library for text manipulation. An object of type *TextView* may register itself with an object of type *TextModel*. The method *text.write(p, ch)* in the *TextModel* will try to keep the position of the caret up to date. Therefore, it will move the caret one position to the right when a character has been inserted in the text left of the caret. In other words, the method *write* of the *TextModel* class will have a statement like

```
if (caretPos ≥ positionAddedCharacter) setCaret(caretPos + 1);
```

However, when a character is typed in the *TextView* object, the caret is moved one position, resulting in a position after the added character. Therefore, because it detects that the actual position of the caret is behind the place where character is written, the *TextModel* object will move the caret again. The two last statements in the method *type* therefore should be exchanged: in that case, the caret will only be moved forward by the *TextView* object, because the *Model* will see that the caret is situated before the added character (and therefore will not move the caret). This is another example of self interference.

Callbacks

A *callback* is a procedure that is passed to a procedure library, to be called at a later point. Without callbacks, it would be possible to use procedures of a procedure library, but "being called" from a procedure library would not be possible. In the case of asynchronous event handling (imagine for instance a library for http requests), it is necessary to have the possibility of callbacks: the alternative is that the program waits until a request is served or is timed out.

A callback reverses the flow of control. Callbacks have an inherent danger of self-interference. It is for that reason that a layered architecture prohibits callbacks. The danger is the possibility that a client queries the state of the library during a callback. This may be done at a moment in which the state of the library is temporarily invalid (does violate the invariant that is specified in the contract), because the library is, at that moment, executing a procedure.

Remedies for self-interference

There are several remedies for this problem of self-interference.

– Require the invariant to be reestablished before relinquishing control. This remedy is usually too strict: method calls to other objects (which can only be done by relinquishing the control to the other object) may be needed to reestablish the invariant.
– Prohibit cyclic call chains, e.g. by prescribing a layered architecture.
– Weaken the notion of invariant: the invariant is guaranteed at the end, but not assumed at the start.
– Use a different specification mechanism, e.g. by specifying permissible histories. Histories are traces of states of a set of variables; specifying permissible histories means restricting the set of possible traces.

8.3 Polymorphism

The concept of polymorphism is used in two ways:

DEFINITION 8.3 *Polymorphism* is the ability of different things to appear the same in a certain context.

DEFINITION 8.4 *Polymorphism* is the ability to appear in several different forms.

Polymorphism is most often used as in the first definition. An interface for instance, can have many different implementations. An example of the second definition is the fact that an object may implement several different interfaces, and thus appear in different forms.
With respect to contracts and invariants, the notion of *substitutability* is important.

DEFINITION 8.5 *B* is substitutable for *A* if *B* can safely be used wherever *A* is required.

This means the following for the relation between contracts:

```
for every operation A.f there is a corresponding B.f such that
pre of A.f ⇒ pre of B.f
and
post of B.f ⇒ post of A.f
```

In other words, *B* can be used where *A* is expected when the precondition of *A.f* implies the precondition of *B.f* and when the postcondition of *B.f* implies the postcondition of *A.f*.
This is easy to understand: before the operation is called, the precondition must be true. If the precondition of *A.f* is true, and this precondition implies the precondition of *B.f*, than it is obvious that *B.f* may be used wherever *A.f* is expected.
After the execution of the operation, the postcondition of *A.f* must be true. If the postcondition of *B.f* implies the postcondition of *A.f*, it is obvious that *B.f* may be used: *A.f* will be true after the execution of *B.f*.

```
interface TextModel
  ...
   void write(int pos, char ch);
   // frame len, txt
   // pre len < max and 0 ≤ pos ≤ len
   // post len = old len + 1 and Ě
   ...


class MyTextModel implements TextModel
  ...
   void write(int pos, char ch)
   // frame len, txt
   // pre len < max and 0 ≤ pos ≤ max
   ...
```

In *TextModel*, the precondition that *len* is smaller than *max* and that *pos* is smaller than or equal to *len* implies that *pos* is smaller than *max*. In other words, with respect to the precondition, *MyTextModel* may be used were *TextModel* is expected. Note that substitutability does not say anything about implementation; it only states things about the contract of the interface.

8.3.1 Contract verification

Ideally, contracts would be explicit and formal. Ideally, it would be possible to have a tool check an implementation and client requests against the contract. However, it is difficult and expensive to write explicit and formal contracts, and an efficient general-purpose verifier is not feasible. What is possible is a tool to support theorem proving, whereby the user proposes logical deduction from a logical statement, and the tool verifies the deduction. However, using such a tool is expensive, as it requires manual assistance. Therefore, developers have to try to avoid errors without the help of contract verification. Some errors can be avoided by a language or by a compiler.

Memory errors

A *memory error* occurs when a program reads a memory cell based on a wrong assumption of what the cell contains. An example is a cell that holds a character, and is read as though it holds a short. Memory errors can affect unrelated parts of programs, are notoriously hard to track down, and can be arbitrarily destructive.

A *type system* can prevent memory errors automatically. A type groups all values of related semantics. A *type* is characterized by an *interface*: a set of operations with signatures.

DEFINITION 8.6 *B* is a *subtype* of *A* (B ⊆ A) means that all operations on *A* are valid on *B*.

This is a transitive relation. In other words, if *C* is a subtype of *B*, and *B* is a subtype of *A*, *C* is a subtype of *A*. Such a type system is supported by many languages. Note that pre- and postconditions are not considered, so substitutability is not guaranteed. To check whether a certain type is a subtype of another type, we have to inspect their interfaces:

```
Let method A.f have an input parameter of type TAI
            and return a value of type TAO.
Let method B.f have an input parameter of type TBI
            and return  a value of type TBO.
For B ⊆ A to be valid, we must have:
   a weaker precondition, so TAI ⊆ TBI (contravariant)
   a stronger postcondition, so TBO ⊆ TAO (covariant)
```

Covariant

The factory method pattern is an example of using the *covariant*: suppose we have a class *CarFactory* with a factory method that creates objects of type *Car*, then objects of a subtype of *CarFactory* may create objects of a subtype of *Car* (for instance a *TruckFactory* that creates a *Truck*).

Contravariant

Now suppose that a CarFactory needs an object of class *Euro* as an input parameter for its method *createCar*, and TruckFactory needs an object of class *Money* as an input parameter for the method *createCar*. When *Euro* is a subtype of *Money*, *TruckFactory* is a subtype of *CarFactory*. When *CarFactory* would need *Money* and *TruckFactory* would need *Euro*, we could have a *Buyer* who uses an object of *Roebel* to pay for a *Truck*, because *Buyer* only knows he is buying from a factory of type *CarFactory*.

Contract changes

A *syntactic change* of a contract means a change in the interface. A *semantic change* of a contract means that the specification of the contract changes.
The *fragile base class problem* is about the problems experienced by derived classes if an ancestor's contract changes. It is prevented by making published contracts immutable (this is for instance forced by Microsoft COM).

8.3.2 Other forms of polymorphism

Overloading

Another form of polymorphism is *overloading* or *ad-hoc polymorphism*: grouping otherwise unrelated operations under the same name. Overloading in Java means: unrelated methods may have the same name if their signatures differ. With overloading, one name (of a method or of an operator) points to different implementations. The choice for the implementation of such a method or operator is based on the types of the parameters or operands.
An example of operator overloading in C^{++} is:

```
Complex operator+(const Complex& x, const Complex& y)

return Complex(x.re() + y.re(), x.im() + y.im());


Complex p(2, 3), i(0, 1), q;
q = p + i
```

Here, the operator + that works with two integers is overloaded with summation of two complex numbers (and is overloaded on other numeric types as well).

Parametric polymorphism

Parametric polymorphism is: letting the same implementation serve several types. An example is formed by templates in C^{++}. Templates in C^{++} lead to code explosion: the preprocessor generates explicit source code for every instantiation. Also, templates themselves cannot be statically type checked.
Another example is formed by *Java generics* (introduced in J2SE 5.0, 2005). In the case of Java generics, there is no code duplication at any level, and there is full static type-checking.
Bounded polymorphism combines parametric and subtype polymorphism. An example is formed by bounded wildcards in Java:

```
public void printBirthdays (List<? extends Person> persons)
```

Here, any *List* may be used that holds elements that are a subtype of *Person*.

8.4 Inheritance

The concept of *inheritance* has three facets, that are often not clearly distinguished:

1. *Subclassing* (implementation inheritance)
2. *Subtyping* (interface inheritance)
3. *Substitutability* (with respect to arguments of methods)

Programming languages have different inheritance concepts.

- Java has (1) (extending a class), (2) (implementing an interface or extending a class) and (3) (the result of a method may be a subtype of the result of the implementation of a parent class, for that method: Java supports covariance in result types).
- Smalltalk only knows (1) and (2).
- In Eiffel, it is possible to claim, within a subclass, that not all features of the parent class are exported. In that case, inheritance means implementation inheritance (1), without subtyping. However, the Eiffel compiler is able to detect when such a claim is not made. In that case, inheritance also means subtyping (2).

8.4.1 Multiple inheritance

There are two principal ways to support *multiple inheritance*:

- *Multiple interface inheritance* like in Java, where a class may implement multiple interfaces. Multiple interface inheritance makes simultaneous substitutability possible, in mutually unaware contexts (the interfaces that are inherited from do not know about each others existence).
- Mixing implementation fragments like in C^{++}. This is easy if the superclasses come from totally disjoint inheritance graphs (and thus do not share common classes), and name clashes are properly resolved. In that case, the new class simply concatenates the inherited fragment

Diamond inheritance

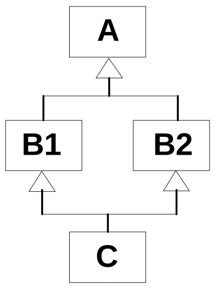

Figure 8.1: Diamond inheritance

Multiple inheritance imposes a problem when two superclasses share a superclass. This is called *diamond inheritance*, see Figure 8.1. Suppose *B1* and *B2* share an attribute *n* defined in *A*, but *B1* uses this to code a boolean *b* via the abstraction invariant

```
b = (n > 0)
```

while *B2* uses *n* to code a boolean *c* via the abstraction invariant

```
c = (2 | n)
```

Then operations on *B1* are only specified in their effects on the sign of *n*, and what their effect is in terms of the abstraction of *B2* cannot be deduced without looking at their implementation in terms of *n*. Hence encapsulation is broken.

So, the question in this case is: do B1 and B2 each get their own copy of the state of A?

- If no, this breaks encapsulation, like in the example above.

- If yes, C may become inconsistent.

An example is: class *A* is a *Car*. Class *B1* is a *Truck* and *B2* is a *FamilyCar*. Class *C* is a *Van*. Now suppose that *Car* has an attribute *fuel*. *Truck* has an attribute *empty* with invariant

```
empty = (fuel = 0)
```

FamilyCar has a boolean *onReserve* with invariant

```
onReserve = (fuel < 15)
```

Suppose *Truck* and *FamilyCar* each have their own copy (*fuel1* and *fuel2*) of attribute *fuel* defined in *Car*. Then the *getFuel()* method defined in *Car* will return *fuel1* if called on a *Truck* object, and *fuel2* if called on a *FamilyCar* object. But a *Van* object has both, and it is not clear whether *fuel1* or *fuel2* should be returned, and class variants for *Van* may get inconsistent.

When *Truck* and *FamilyCar* do not have their own copy of attribute *fuel*, and *FamilyCar* has a method *beOnTimeAtAirport*, with postcondition

```
fuel >= 0
```

we have to check the implementation of that method to see what influence it has on the class invariant for *empty*.

Mixins

Mixins are a particular style of multiple inheritance. A class inherits an interface from one superclass, and an implementation from several classes, called mixins. Mixins are used to build e.g. GUI's. Every mixin concentrates on a single aspect, e.g. window borders. Independence is necessary but not enforced.

Traits

Traits (introduced in the Scala programming language) combine advatages of noth interfaces and mixins. A class may use several traits, and, like Java interfaces, traits extend the interface of the class using them. Traits differ from interfaces in that methods declared in a trait may be both without and with an implementation. Diamond inheritance is prevented by making the order in which traits are used important. Method implementations of last trait mentioned override method implementations of traits earlier mentioned.

```
class CollegeStudent extends Person with Student with Worker
```

In this example, method implementations of trait Worker will be used, in case there are method implementations for a method with the same name in trait Student and traid Worker.

8.4.2 The fragile base class problem

The *syntactic fragile base class problem* is about binary compatibility of compiled classes with new binary releases of superclasses. If the interface of such a newly released base class has changed, there is no binary compatibility. In the case that such an incompatibility is prohibited, it is called *release-to-release binary compatibility* (RRBC).

Some changes are harmless, but they should not require recompilation of subclasses. Examples of harmless changes are reordering attributes and methods, or moving methods up in the class hierarchy. The question with respect to the *semantical fragile base class problem* is: will a subclass remain valid if the implementation of a superclass changes?

Suppose: superclass A has independent methods f and g. Subclass B overrides f with an implementation that calls g in an intermediate state. Later, $A.g$ is replaced with a version that assumes a class invariant to hold at its start. Now $A.g$'s assumption is invalid. This problem is similar to the callback issue.

8.4.3 Disciplined inheritance

Disciplined inheritance means that one imposes restrictions on inheritance, to avoid semantic fragile base class issues. A simple method is to specify interfaces as protected or private.

Another method is the Stata-Guttag method [53]: split state and behavior into groups. Method groups encapsulate part of the state. Method groups form co-dependent cliques, including the necessary attributes. Either all or none of the methods in a group are overridden during inheritance. The groups then are like cooperating classes.

8.5 Delegation

Delegation, also known as object composition, is an alternative to inheritance. Messages are forwarded to so called inner objects (objects which are associated by a composition relation). Delegation supports late composition. The identity of the delegator is remembered.

What a program does is often harder to understand when delegation is used than in the case of inheritance. Delegation works best in standardized settings, such as design patterns: the name of the pattern then helps to understand what is going on.

8.6 Techniques for specification

8.6.1 Eiffel

Eiffel is an interesting language with respect to contracts: it is an object-oriented programming language with full support for preconditions, postconditions and class invariants.

```
class STACK[T]
feature
  count: INTEGER
  capacity: INTEGER
  top: T is
    require not empty
    do ... end
  empty: BOOLEAN is ...
  push(elem: T) is
```

```
    require not full
    do ... end
    ensure
      not empty
      top = elem
      count = old count + 1
    feature NONE -- implementation
  rep: ARRAY[T]
end STACK

invariant
  0 <= count
  count <= capacity
  capacity = rep.capacity
  empty = (count = 0)
  (count > 0) implies (rep.item(count) = top)
```

This Eiffel code shows a class and a class invariant. A `feature` is a combination of attributes and methods; a class may have several features. The statement `feature NONE` means that this feature is invisible for all other objects. A precondition is written using the keyword `require`, and a postcondition with the keyword `ensure`. In this example, the precondition of the method `push` is that the stack is not full, and the postcondition is that the stack is not empty, that the new element is placed on top, and that the number of elements has grown with one. The class invariant in the example states that the number of elements may be zero, and may reach the value of `capacity`. It also states that the method `empty` returns `TRUE` is the number of elements is zero. It also states that, when the stack is not empty, the method `top` returns the top element of the stack.

The explanation of the meaning of the statements in the class invariant shows some problems with assertions in Eiffel:

- Class invariants are also used to characterize the result of functional methods. This is the only way to state such postconditions, because the keyword `ensure` may only be used to specify state changes. This means that method specification gets mixed with the characterization of legal object states. This is a consequence of the Eiffel philosophy not to distinguish between functional methods and read-only public fields.
- There is no reference to the meaning of objects; assertions only specify relations between fields and method results. In our example for instance, there is no possibility to describe that the stack contains a sequence of values.
- No distinction is made between external behavior and internal representation, because postconditions are expressed in state changes.
- Every assertion must be a boolean expression. This means that it is hard or impossible to specify the postcondition of certain methods, for instance, of a method that returns the sum of the values of the elements.

8.6.2 Model-based specifications

The Eiffel mechanism does not distinguish between specification as guidance for implementors and client programmers, specification as basis for correctness proofs, and specification for runtime checking. In Java for instance, such a distinction does exist:

javadoc comments serve as a guidance for implementors and client programmers, assert statements for runtime checking.

An alternative (for guidance and as a basis for correctness proof) is:

– Describe the properties of objects in terms of the problem domain being modeled. In the example: a stack represents a finite sequence in T^*.
– Specify operations in terms of the model only. This means: do not use any attributes. As a result, it is also possible to specify operations of an interface, and the specifications remain valid when you change the representation. In the example:

```
push(elem) ensures stack = [elem] ++ old stack
```

– Specify the implementation in terms of a class invariant. In the example this would be:

$$\forall i : 0 \leq i \leq \textit{capacity} : stack\{i\} = rep.item[i]$$

DISCUSSION QUESTIONS

1. An inherent weakness of the precondition/postcondition style of specification is that it says nothing about the method's behavior in case abnormal termination is impossible (e.g. because of failing connections or incorrect user input). How would you remedy this?

2. Several notions of a class invariant have been introduced (one that holds at every change of control, one that holds upon method completion, one that holds when no method is executing). Can you think of examples showing that in practice all of these occur?

3. The open/closed principle states "software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification" [31]. A class or module should, according to this principle, allow its behaviour to be modified without altering its source code. Relate this principle to the fragile base class problem.

Module 9

Service oriented architectures

INTRODUCTION

Service orientation is a particular strategy for separating concerns and dividing a system into components. Its fundamental characteristic is that every component provides a distinct service that can be used by multiple consumers. Service logic can encompass the logic provided by other services: the point of such a service is that it composes a number of other services into a collective.

The paradigm of service orientation is based on the interaction of businesses in an ideal market: a business should then be able to use services offered by others, and offer services to others. Likewise, service orientation can be used within an organization. An example might be a bank. Imagine that several areas of banking applications will deal with the current balance of an existing customer. More than often, the "get current balance" functionality is repeated in various applications within a banking environment. This gives rise to a redundant programming scenario. The focus should be toward finding this sort of common, reusable functionality and implement it as a service, so that all banking applications can reuse the service as and when necessary.

With minimal interdependencies, the number of services may grow autonomously, within and between organizations. Adherence to standardized interaction conventions make minimal interdependency possible.

In this module, we will discuss the key aspects of services, show you the various standards that are used for service oriented architectures, discuss the concepts of orchestration and choreography, and give an example of a platform for service oriented architecture.

LEARNING OBJECTIVES
After having studied this module you should be able to:

– explain the meaning of the concepts of service-oriented architecture, coordination, orchestration and choreography,
– mention the key aspects of service-oriented architecture,
– describe the purpose of the existing standards concerning service-oriented architecture.

9.1 Aspects of services

9.1.1 Communication

A service can only be used if potential clients are aware of its existence, and have information about how to use it. This is achieved through *service description*.

DEFINITION 9.1 A *service description* contains at least the name of the service, the location of the service, and the data exchange requirements.

Communication between services is done through messages, asynchronously: services communicate over the internet, and the loss of a connection or of a communicating partner should not block the system that offers the service. Messages should be autonomous, which means that they must be outfitted with enough information to self-govern their part of the processing logic.

9.1.2 Key aspects of services

Key aspects of services are:

- *Loose coupling*. As we have already seen, the inherently unreliable communication over the internet must not lead to a blocked system. Furthermore, services should have minimal dependencies. The only requirement is that there should be an awareness of the existence of a service.
- *Service contract*. A *service contract* is a communications agreement, and specifies the details of the service that the service provider is performing. In a service contract, functional and non-functional requirements are described, and so-called policies are specified, stating obligations and responsibilities of both the service provider and the service consumer.
- *Autonomy*. Services should have control over the logic they encapsulate.
- *Abstraction*. Services hide logic from the outside world, except for what is described in the service contract.
- *Reusability*. Obviously, services are meant to be reused over and over again.
- *Composability*. When services are composable, collections of services can be coordinated and assembled for composite services.
- *Statelessness*. To be as reliable as possible, especially when communication is done over the internet, services should minimize retaining information specific to an activity. Statelessness is important for loose coupling. Any state that should be maintained during a transaction should be maintained by the client, and supplied by the client in the next step of the process, if there is a next step.
- *Discoverability*. A service should be described in such a way that its description can be found and assessed via available discovery mechanisms.

9.1.3 SOA and web services

DEFINITION 9.2 *Service-oriented architecture* (SOA) means: an architecture entirely composed of services.

This paradigm does not care about implementations, and can be realized with any suitable technology platform. Web services, as they have developed over the last few years, are a very successful technology platform for the support of a service-oriented architecture. Web services include technology standards such as SOAP, WSDL, XML and XML Schema.

But keep in mind that using web services does not automatically make an architecture service-oriented. What makes an architecture service-oriented is that it may be viewed as a configuration of components, with each component offering functionality by providing a service.

Quality of service

For a service-oriented architecture to be able to implement enterprise-level functionality as safely as classical distributed architectures, it is required that tasks are carried out in a secure manner, protecting the contents of a message and access to individual services.

Tasks should both be carried out *reliably* so that message delivery or notification of failed delivery can be guaranteed, and *securely* so that messages cannot be seen or changed by a third party. The performance overhead imposed by SOAP messaging and XML content processing should not inhibit the *timely* execution of a task. And, very important, the *integrity* of transactions should be guaranteed; upon failure, exception logic should be executed.

The use of web services per se does not provide any of these guarantees.

Open standards

An advantage of web services is that data exchange is governed by *open standards*. Examples of these standards are SOAP, WSDL, XML and XML Schema, which have been discussed in a previous module.

The standards are developed for the purpose that messages can be completely self-contained. To communicate using web services according to these standards, services require nothing more than a knowledge of each other's service descriptions: there is no need for shared dependencies such as type systems.

The role of proprietary technology is limited to the implementation and hosting of the application logic encapsulated by a service: a service-oriented architecture using web services supports vendor diversity. Proprietary development environments are used to create a non-proprietary service interface layer (think for instance of the support for the use of SOAP in a J2EE environment).

9.1.4 Adopting SOA

Not all features necessary for enterprise-level computing are fully available for service-oriented architectures at this moment. Standards organizations and software vendors have proposed many specifications to address a variety of supplementary extensions.

The next generation of development tools and application servers promises to support a great deal of these new technologies.

Benefits of adopting SOA are:

- There is an intrinsic interoperability: cross-application integration is simplified.
- Services are reusable.
- Integration of legacy environments is relatively easy: they can participate via adapters.
- There is a standardized XML data representation. XML is self-descriptive via XML Schema.
- Service-oriented architectures increase organizational agility: reorganizations and merging different organizations for instance are more easily supported. An environment of loosely coupled services is more adaptable: it will be easier to support new business rules or different requirements. Business logic and technology are

abstracted into separate service layers, which means that it will be easier to introduce new technology.

However, these benefits only manifest themselves in the long run, as the use of service-oriented principles becomes established.

There are also *pitfalls* in adopting SOA:

- One pitfall is to build a service-oriented architecture like traditional distributed architectures. Such an approach will lead to fine-grained RPC-style (remote procedure call) message exchanges, and to the creation of non-composable services.
- Another pitfall is not to standardize. This will lead to disparate schemas, representing similar information.
- It is also a pitfall not to understand the performance requirements of service-oriented architectures. Service orientation introduces several extra layers of data processing with associated performance overhead compared to traditional RPC. There is accelerating technology available, e.g. XOP (XML-binary Optimized Packaging) and MTOM (Message Transmission Optimization Mechanism) (both W3C standards).
- A fourth pitfall is not to keep in touch with platform and standard development: service orientation is an area in development.

9.2 Standards and technologies

9.2.1 History of web service technologies

XML (eXtensible Markup Language) was developed in 1998, by the W3C (see below). One of its goals was to facilitate internet e-business. Additional techniques are XSLT (eXtensible Stylesheet Language Transformations), from 1999, and XSD (XML Schema Definition Language), from 2001.

SOAP (originally: Simple Object Access Protocol) was developed in 1998, with support from Microsoft. It has become a W3C standard since 2000. Soap is a message protocol that is human-readable, as opposed to binary formats. It works well with network firewalls, as opposed to for instance DCOM (from Microsoft). SOAP was originally intended only for RPC (as is reflected in its name); not for messaging. As of version 1.2 (in 2003), the emphasis has shifted to messaging. Its name was considered misleading, and therefore, SOAP is no longer an acronym.

WSDL (Web Service Definition Language) was developed in 2001, with support from Ariba, IBM and Microsoft. It has been submitted for standardization to the W3C.

UDDI (Universal Description, Discovery and Integration) was developed in 2000, by `UDDI.org` and OASIS (see below). It is not yet universally adopted.

9.2.2 Standards and collaboration organizations

Important standards organizations are:

- The *W3C* (World Wide Web Consortium) was founded in 1994, by Tim Berners-Lee. Some standards of the W3C are HTML, XML, XSD, XSLT and SOAP.
- *OASIS* (Organization for the Advancement of Structured Information Standards) was originally named SGML Open, which was founded in 1993. In 1998, the name has changed into OASIS. Some standards of OASIS are WS-BPEL (Web Services Business Process Execution Language), ebXML (Electronic Business XML),

UDDI, SAML (Security Assertion Markup Language), XACML (eXtensible Access Control Markup Language) and WS-Security (Web Services Security). The focus of OASIS is on vertical industry support (which means support from all organizations within a certain domain, as opposed to many different domains).

- The *WS-I* (Web Services Interoperability) Organization was established in 2002. WS-I recommends which standards should be collectively used to promote optimum interoperability.
- The *OSOA* (Open Service-Oriented Architecture) Collaboration is an informal alliance of companies aiming to define a language-neutral programming model for service-oriented architectures. It is not a standards organization: the goal is to rapidly develop specifications for later standardization. It was founded November 2005, and the first completed specifications will probably be there in 2007. Among the participants are BEA, IBM, Oracle, SAP, Sun Microsystems, Tibco and RedHat; Microsoft is not among them.

Two of the projects of the OSOA Collaboration are the SCA (Service Component Architecture) project, and the SDO (Service Data Objects) project. SCA is a model for building service-oriented systems (not necessarily based on web services). SDO is a unified model for data handling (not necessarily from XML data sources).

9.3 Coordination of services

9.3.1 The WS-Coordination standard

The central theme in the *WS-Coordination standard* is that of coordination context information.

DEFINITION 9.3 In general terms, *coordination* is the act of a coordinator disseminating information to a number of participants for some domain-specific reason.

This reason could be to reach consensus on a decision like a distributed transaction protocol, or simply to guarantee that all participants obtain a specific message.

When parties are being coordinated, *coordination!context* is propagated to tie together operations which are logically part of the same coordinated work or activity. Examples of data held within a coordination context are a unique identifier that represents the activity, an expiration value, and coordination type information (such as an atomic transaction or a business activity).

The WS-Coordination standard introduces a generic service around the notion of coordination context information. This service controls a composition of:

- An *activation service* which is responsible for the creation of a new context and for associating this context to a particular activity.
- A *registration service* which allows participating services to use context information received from the activation service to register for a supported context protocol.
- *Protocol-specific services*. Protocols represent unique variations of coordination types, and consist of a collection of specific behaviors and rules. The WS-Coordination framework is extensible with respect to coordination types and protocols.
- A *coordination service*: the controller service of the composition.

9.3.2 The WS-AtomicTransaction standard

WS-AtomicTransaction is a coordination type, i.e. an extension created for use with the WS-Coordination context management framework. It provides three primary transaction protocols:

- The *Completion protocol*, typically used to initiate the commit or abort states of the transaction,
- the *Durable 2PC protocol* for which services representing permanent data repositories should register, and
- the *Volatile 2PC protocol* to be used by services managing non-persistent data.

The abbreviation 2PC means two-phase commit.

The process for an atomic transaction is as follows:

- The atomic transaction coordinator decides the outcome of a transaction, based on feedback from all participants.
- In the first phase (or *prepare phase*) all participants are notified and asked to issue a vote (either "commit" or "abort").
- The second phase (or *commit phase*) has two possible outcomes: either the transaction is successful and all changes are committed if all votes are received and all participants voted to commit; or all changes are rolled back.

9.4 Orchestration

DEFINITION 9.4 *Orchestration* is a centrally controlled collection of workflow logic that takes care of the interoperability between various processes or legacy environments.

Orchestration can be seen as the ability to control how information flows and services interact to form solutions. In essence, orchestration is a meta application layer over and above existing services and data that binds systems and services together. It was originally developed for middleware for enterprise application integration, but is very usable for the coordination of services in a service-oriented architecture.

Orchestration is often implemented by a *hub-and-spoke model* and a central orchestration engine. With such an approach, workflow is more easily maintained than when embedded in individual solution components.

In service-oriented architectures, orchestrations themselves exist as services that represent and express business logic.

9.4.1 The WS-BPEL standard

WS-BPEL is the primary industry specification that standardizes orchestration. The standard was proposed in 2002, jointly by IBM, Microsoft, and BEA. It was originally called BPEL4WS, and is now a proposed standard from OASIS.

WS-BPEL is an XML based programming language to describe high level business processes. A business process is a term used to describe the interaction between two businesses or two elements in some business. An example of this might be company A purchasing something from company B. BPEL allows this interaction to be described easily and thoroughly such that company B can provide a Web Service and company A can use it with a minimum of compatibility issues.

Using BPEL you can write a program that uses a web service to fetch the weather forecast for your postcode (zip code), check whether it will be raining this weekend and use another web service to send you an SMS message suggesting that you go shopping (if it's raining) or go to the beach (if it's sunny) [32].

```
<?xml version="1.0" encoding="UTF-8"?>
<process
 xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
 xmlns:print="http://www.eclipse.org/tptp/choreography/2004/engine/Print"
 <import importType="http://schemas.xmlsoap.org/wsdl/"
  location="../../test_bucket/service_libraries/tptp_EnginePrinterPort.wsdl"
  namespace="http://www.eclipse.org/tptp/choreography/2004/engine/Print" />
 <partnerLinks>
  <partnerLink name="printService"
    partnerLinkType="print:printLink"
    partnerRole="printService"/>
 </partnerLinks>
 <variables>
  <variable name="hello_world"
            messageType="print:PrintMessage" />
 </variables>
 <assign>
  <copy>
   <from><literal>Hello World</literal></from>
   <to>\$hello_world.value</to>
  </copy>
 </assign>
 <invoke partnerLink="printService" operation="print"
    inputvariable="hello_world" />
</process>
```

Figure 9.1: Hello World in WS-BPEL

WSDL is used to describe individual services; WS-BPEL is used to describe processes that are supported by services.

BPMN (Business Process Modeling Notation) is developed by the Object Management Group in 2006. It is a graphical notation for modeling an organization's internal business processes, similar in intention and scope to UML. There is a mapping to WS-BPEL.

9.5 Choreography

Orchestration expresses organization-specific workflow; *choreography* is not owned by a single entity and acts as a community interchange pattern for collaborative purposes. *WS-CDL* (Web Services Choreography Description Language) organizes information exchange between multiple organizations in order to make collaboration possible. None of the organizations controls the collaboration logic.

Choreography is defined in terms of *relationships* (potential exchanges) and *channels* (defining the nature of the messages exchanged). Every relationship consists of exactly two roles. Relationships determine who can talk to whom; channels establish the nature of the conversation. Channel information can be passed around in a message, facilitating dynamic discovery of potential collaborators.

WS-BPEL (by OASIS, for orchestration) and WS-CDL (by W3C, for choreography) overlap somewhat in functionality.

9.6 Security and reliability

9.6.1 The WS-ReliableMessaging standard

The *WS-ReliableMessaging standard* is developed by IBM, Microsoft, BEA, and Tibco in 2005. It specifies a protocol allowing messages to be delivered reliably between distributed applications, in the presence of software component, system, or network failures. The protocol is described in a technology-independent manner, allowing it to be implemented using different network transport technologies.

The protocol provides a framework guaranteeing that service providers will be notified of the success or failure of message transmissions, and that a message sent with specific sequence-related rules will arrive as intended or generate a failure condition. The protocol is based on acknowledgments from destination to source.

9.6.2 WS-Security

The *WS-Security framework* provides extensions that can be used to implement message-level security measures, to protect message content during transport and during processing by service intermediaries.

WS-Security establishes a standardized header block for SOAP messages that expresses their origin or owner (called a token). The token contains elements like Username en Password. Binary data, such as certificates, can be represented in an encoded format.

9.6.3 Single sign-on

The challenge of *single sign-on* is to propagate the authentication and authorization information for a service requestor across multiple services behind the initial service provider. *Authentication* means proving an identity (for instance using a username and password); *authorization* means checking what the requestor, once authenticated, is allowed to do.

There are several technologies for single sign-on:

- Using *SAML* (Security Assertion Markup Language), the point of contact for a service requestor can act as issuing authority. The issuing authority assures other services that the requestor has attained the proper level of clearance. An identity provider stores identities, and allows users to log in. Service providers (web applications) may use such an identity provider for authentication puposes.
- OpenID is an open standard that describes how users can be authenticated in a decentralized manner. A user chooses a URL where he or she provides a username and a password, and that URL will be used on other sites to allow the user to log in.
- OAuth is an open standard for authorization, often used in social media.
- *XACML* (XML Access Control Markup Language) is a language to specify and standardize policy management and access decisions. It is developed by OASIS.

9.7 Platforms for service-oriented architectures

For a platform (consisting of a programming language, APIs, and a runtime environment) to be SOA-capable it should provide:

- the ability to partition software into components capable of communicating with each other within and across instances of the runtime environment,
- the ability to encapsulate and expose components through web service standards (WSDL, SOAP, UDDI, WS-*). The platform should thus be able to supply, discover and interpret WSDL definitions, receive and transmit SOAP messages, assemble and process SOAP header blocks, and validate, parse and transform the payload of a SOAP message.

9.7.1 SOA support in J2EE

Support for service-oriented architectures is offered within the J2EE platform in the form of:

- The Java API for XML Processing (*JAXP*). This API is used to process XML document content using a number of available parsers. It supports the transformation and validation of XML documents using XSLT stylesheets and XSD schemas.
- The Java API for XML-based RPC (*JAX-RPC*). This API offers support for SOAP processing. It supports both RPC-type and asynchronous exchanges and one-way transmissions. Web services are implemented as a stateless servlet.
- The Java API for XML Registries (*JAXR*). JAXR is a standard interface for accessing business and service registries. It supports UDDI.
- The Java API for XML Messaging (*JAXM*). JAXM is an asynchronous messaging API that can be used for one-way and broadcast message transmissions.
- The SOAP with Attachments API for Java (*SAAJ*).
- The Java Architecture for XML Bindings (*JAXB*). JAXB generates Java classes from XSD schemas.
- The Java Messaging Service API (*JMS*). This is a messaging protocol for traditional messaging middleware solutions, providing reliable delivery features not found in typical HTTP communication.
- Vendor-specific extensions, e.g. IBM Emerging Technologies Toolkit: prototype implementations of a number of WS-* extensions.

9.8 RESTful webservices

REST (Representational State Transfer) [18] is an architecture; not a standard. Based on REST-principles, lightweight webservices are possible. Such a web service is located by a URI, there is a MIME type (such as JSON or XML) associated with the data that are sent, and communication is stateless, using HTTP. An example of a RESTful webservice is an Atom or RSS feed: a website or webpage offers summaries of news through a web service. Data sent through such a webservice adheres to one of the standards for RSS (Atom or RSS).

DISCUSSION QUESTIONS

1. It is often claimed that SOA is the next phase in the evolution of business automation, just as mainframe systems were succeeded by client-server applications, and these in turn evolved into classical distributed systems. Others claim that SOA is just a marketing term used to re-brand distributed computing with web services. Whose side are you on and why?

2. What impact does the introduction of SOA have on an organization's capabilities? How would you convince a non-technical manager of its benefits?

3. Can you think of applications where the introduction of SOA would be detrimental and should be resisted?

Bibliography

[1] G. Abowd, L. Bass, P. Clements, R. Kazman, L. Northrop, and A. Zaremski. Recommended best industrial practice for software architecture evaluation. Technical Report CMU/SEI-96-TR-025, Carnegie Mellon Software Engineering Institute, 1997.

[2] C. Alexander. *The Timeless Way of Building*. Oxford University Press, 1979.

[3] C. Alexander and S. Ishikawa. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

[4] Apache Software Foundation. Struts. `http://struts.apache.org/`.

[5] F. Bachmann, L. Bass, and R. Nord. Understanding architectural patterns in terms of tactics and models. `http://www.sei.cmu.edu/library/abstracts/news-at-sei/architect200708.cfm`, 2007.

[6] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice (2nd ed.)*. Addison Wesley, 2003.

[7] A. Birrel and B. Nelson. Implementing Remote Procedure Calls. Technical Report CSL-83-7, Xerox, 1983.

[8] I. T. Bowman, R. C. Holt, and N. V. Brewster. Linux as a case study: its extracted software architecture. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 555–563, New York, NY, USA, 1999. ACM.

[9] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0, w3c recommendation 10-february-1998. Technical Report REC-xml-19980210, World Wide Web Consortium W3C, 1998.

[10] J. Burg. Business modeling and requirements engineering. dictaat VU, 2005.

[11] Carnegie Mellon Software Engineering Institute. Software product lines. `http://www.sei.cmu.edu/productlines/`, 2006.

[12] P. Clements, F. Bachmann, L. Bass, D. Darlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures, Views and Beyond*. The SEI Series in Software Engineering. Addison-Wesley, 2003.

[13] P. Clements, R. Kazman, and M. Klein. *Evaluating Software Architectures, Methods and Case Studies*. The SEI Series in Software Engineering. Addison-Wesley, 2002.

[14] Eclipse Open Source. JFace. `http://dev.eclipse.org/viewcvs/index.cgi/~checkout~/platform-ui-home/main.html`.

[15] Eclipse Open Source. SWT, the Standard Widget Toolkit. `http://www.eclipse.org/swt/`.

[16] Eiffel Software. Frequently asked questions about the Eiffel language. http://www.eiffel.com/developers/faqs/eiffel-language.html.

[17] D. C. Fallside and P. Walmsley. XML Schema Part 0: Primer Second Edition, W3C Recommendation 28 October 2004. http://www.w3.org/TR/xmlschema-0/, 2004.

[18] R. Fielding and R. Taylor. Principled design of the modern Web architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2):115–150, 2002.

[19] M. Fowler. *Patterns of Enterprise Application Architecture*. Pearson Education, Boston, 2003.

[20] E. Gamma, R. Helm, R. Johnsons, and J. Vlissides. *Design patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[21] J. Garrett. Ajax: A new approach to web applications. http://www.adaptivepath.com/publications/essays/archives/000385.php, 2005.

[22] C. B. Hissam, Scott A. Weinstock. Open source software: The other commercial software. In *Making Sense of the Bazaar: 1st Workshop on Open Source Software Engineering*, 2001.

[23] C. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Addison-Wesley, 1985.

[24] G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. The Addison-Wesley Signature Series. Addison-Wesley, 2004.

[25] International Standard Organization. ISO/IEC-9126-1:2001, information technology – product quality – part1: Quality. Technical report, International Standard Organization, 2001.

[26] G. Kotonya and I. Sommerville. *Requirements Engineering: Processes and Techniques*. Worldwide Series in Computer Science. John Wiley and Sons, 1998.

[27] P. Kruchten. Architectural blueprints: the 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

[28] A. v. Lamsweerde. *Requirements Engineering: from System Goals to UML Models to SoftwareSspecifications*. John Wiley and Sons, 2009.

[29] S. Lilly. How to avoid use-case pitfalls. *Software development*, 8(1):40–44, January 2000.

[30] L. Maciaszek. *Requirements Analysis and System Design. 2nd ed.* Addison Wesley, 2005.

[31] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[32] A. Miguel. WS-BPEL 2.0 Tutorial, 2005.

[33] Object Management Group. Introduction to omg's unified modeling language (uml). http://www.omg.org/gettingstarted/what_is_uml.htm, 2005.

[34] Object Management Group. UML 2.0 OCL Specification. http://www.omg.org/docs/ptc/03-10-14.pdf, 2006.

[35] Object Management Group. UML Resource Page, UML2.0. `http://www.omg.org/uml/`, 2006.

[36] M. Odersky. The Scala experiment – can we provide better language support for component systems? In *Proc. ACM Symposium on Principles of Programming Languages*, pages 166–167, 2006.

[37] OpenOffice. Openoffice architecture. http://wiki.services.openoffice.org/wiki/Architecture.

[38] S. S. Owicky and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.

[39] D. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–105, December 1972.

[40] Prismtech. Opensplice. `http://www.prismtech.com/`.

[41] D. Riehle. The economic motivation of open source software: Stakeholder perspectives. *IEEE Computer*, 40(4):25–32, April 2007.

[42] D. T. Ross. Structured analysis (sa): A language for communicating ideas. *IEEE Transactions on Software Engineering*, 3(1):16–34, 1977.

[43] N. Rozanski and E. Woods. *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley Professional, 2005.

[44] N. Rozanski and E. Woods. Introduction to perspectives. `http://www.viewpoints-and-perspectives.info/index.php?page=persp-intro`, 2009.

[45] SERC. Quint2, the extended iso model of software quality. `http://www.serc.nl/quint-book/`.

[46] M. Shaw. Larger scale systems require higher-level abstractions. In *Proceedings of the 5th international workshop on Software specification and design*, pages 143–146, 1989.

[47] M. Shaw and P. Clements. A field guide to boxology: Preliminary classification of architectural styles for software systems. In *Proceedings of the 21st International Computer Software and Applications Conference*, pages 6–13, 1997.

[48] M. Shaw and D. Garlan. *An Introduction to Software Architecture*, volume 2 of *Series on Software Engineering and Knowledge Engineering*, pages 1–39. World Scientific Publishing Company, Singapore, 1993.

[49] I. C. Society. Standard glossary of software engineering terminology (ieee std610.12-1990), 1990.

[50] Software Engineering Standards Committee of the IEEE Computer Society. IEEE Recommended Practice for Architectural Descriptions of Software-Intensive Systems, Std 1471-2000, 2000.

[51] Software Verification Research Centre, University of Queensland. The Object-Z Homepage. `http://www.itee.uq.edu.au/~smith/objectz.html`.

[52] A. Sotira. What is Gnutella? `http://www.gnutella.com/news/4210`.

[53] R. Stata and J. V. Guttag. Modular reasoning in the presence of subclassing. In *Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pages 200–214, 1995.

[54] C. Szyperski and W. Weck. Do we need inheritance? In *Workshop on Composability Issues in Object-Orientation (at ECOOP'96)*, 1996.

[55] The Zachman Institute. Site of the Zachman Institute for Framework Advancement, ZIFA. `http://www.zifa.com/`, 2006.

[56] E. Yourdon. Just enough structured analysis, chapter 22, moving into design. `http://www.yourdon.com/strucanalysis/chapters/ch22.html`, 2006.

[57] E. Yourdon. Just enough structured analysis, chapter 9, data flow diagrams. `http://www.yourdon.com/strucanalysis/chapters/ch9.html`, 2006.