Software Architecture Design

Dr Fritz Solms
fritz@solms.co.za
http://www.solms.co.za

November 14, 2014
Contents

1 Introduction to software architecture  
1.1 Is architecture a common concept? 9
1.2 The purpose of architecture 9
1.3 Questions addressed by software architecture 10
1.4 What is software architecture? 10
  1.4.1 Suggestions? 10
  1.4.2 Definitions of software architecture 10
  1.4.3 Reference architectures 13
  1.4.4 Definition of software architecture 17
  1.4.5 Components of software architecture 17
  1.4.6 Typical responsibilities of architecture 17
  1.4.7 Pure application code 18
  1.4.8 Architecture neutral design and implementation 18
  1.4.9 Interplay between architecture and application design 18
  1.4.10 Architecture as a matter of perspective 18
1.5 The bigger picture 19
1.6 Wish list 19
1.7 Software architecture in the context of MDD 20
1.8 Responsibilities of software architecture 20

2 Structural patterns 23
2.1 What are structural patterns 23
2.2 Why use structural patterns? 23
2.3 Layering 23
  2.3.1 Uses of the layering pattern 24
  2.3.2 Advantages and disadvantages of layering 25
2.4 Microkernel 25
  2.4.1 Uses of the microkernel pattern 26
  2.4.2 Benefits and challenges of the microkernel pattern 26
2.5 Blackboard 27
  2.5.1 Uses of the blackboard pattern 28
  2.5.2 Benefits and concerns around the blackboard pattern 28
2.6 Master-Slave 29
  2.6.1 Uses of the master-slave pattern 30
  2.6.2 Benefits and concerns 30
2.7 Hierarchical 30
  2.7.1 Uses of the hierarchical pattern 31
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.7.2 Benefits and challenges of the hierarchical pattern</td>
<td>31</td>
</tr>
<tr>
<td>2.8 Pipes and filters</td>
<td>32</td>
</tr>
<tr>
<td>2.8.1 The pipes &amp; filters pattern</td>
<td>32</td>
</tr>
<tr>
<td>2.9 Model-View-Controller (MVC)</td>
<td>33</td>
</tr>
<tr>
<td>2.9.1 Uses of the MVC pattern</td>
<td>34</td>
</tr>
<tr>
<td>2.9.2 Benefits and concerns around the MVC pattern</td>
<td>34</td>
</tr>
<tr>
<td>2.10 Patterns and quality attributes</td>
<td>35</td>
</tr>
<tr>
<td>2.11 Exercises</td>
<td>35</td>
</tr>
<tr>
<td>3 Architectural tactics</td>
<td>37</td>
</tr>
<tr>
<td>3.1 What is an architectural tactic?</td>
<td>37</td>
</tr>
<tr>
<td>3.2 Why use architectural tactics?</td>
<td>37</td>
</tr>
<tr>
<td>3.3 Some examples of architectural decisions</td>
<td>38</td>
</tr>
<tr>
<td>3.4 Scalability and Performance</td>
<td>38</td>
</tr>
<tr>
<td>3.5 Reliability</td>
<td>39</td>
</tr>
<tr>
<td>3.6 Security</td>
<td>40</td>
</tr>
<tr>
<td>3.7 Modifiability and extensibility</td>
<td>41</td>
</tr>
<tr>
<td>3.8 Accessibility and integrability</td>
<td>42</td>
</tr>
<tr>
<td>3.9 Tactics and quality attributes</td>
<td>44</td>
</tr>
<tr>
<td>3.10 Exercises</td>
<td>45</td>
</tr>
<tr>
<td>4 Integration architecture</td>
<td>47</td>
</tr>
<tr>
<td>4.1 Integration challenges</td>
<td>47</td>
</tr>
<tr>
<td>4.1.1 Accessibility challenges</td>
<td>47</td>
</tr>
<tr>
<td>4.1.2 Infrastructure constraints</td>
<td>47</td>
</tr>
<tr>
<td>4.1.3 Providing quality attributes across integration channels</td>
<td>48</td>
</tr>
<tr>
<td>4.1.4 Lack of contracts</td>
<td>48</td>
</tr>
<tr>
<td>4.1.5 Integration complexity</td>
<td>48</td>
</tr>
<tr>
<td>4.2 Integration concepts</td>
<td>48</td>
</tr>
<tr>
<td>4.2.1 Component or service publication</td>
<td>48</td>
</tr>
<tr>
<td>4.2.2 Communication protocols</td>
<td>49</td>
</tr>
<tr>
<td>4.2.3 Integration approaches</td>
<td>49</td>
</tr>
<tr>
<td>4.2.4 Integration mechanisms</td>
<td>50</td>
</tr>
<tr>
<td>4.3 Integration patterns</td>
<td>50</td>
</tr>
<tr>
<td>4.3.1 Messaging benefits</td>
<td>51</td>
</tr>
<tr>
<td>4.3.2 Why</td>
<td>51</td>
</tr>
<tr>
<td>4.3.3 Message channels</td>
<td>51</td>
</tr>
<tr>
<td>4.3.4 Message construction patterns</td>
<td>55</td>
</tr>
<tr>
<td>4.3.5 Routing patterns</td>
<td>59</td>
</tr>
<tr>
<td>4.3.6 Transformation patterns</td>
<td>64</td>
</tr>
<tr>
<td>4.3.7 End-point patterns</td>
<td>67</td>
</tr>
<tr>
<td>4.3.8 System management patterns</td>
<td>71</td>
</tr>
<tr>
<td>4.4 Best practices</td>
<td>74</td>
</tr>
<tr>
<td>4.4.1 Some general best practices around integration</td>
<td>74</td>
</tr>
<tr>
<td>4.4.2 Best practices for request-based integration</td>
<td>74</td>
</tr>
<tr>
<td>4.4.3 Best practices for document-based integration</td>
<td>74</td>
</tr>
<tr>
<td>4.4.4 Best practices for integration infrastructure</td>
<td>75</td>
</tr>
<tr>
<td>4.5 Exercises</td>
<td>75</td>
</tr>
</tbody>
</table>
CONTENTS

6.2 Systematic Method for Architecture Design (SyMAD) .................................. 111
  6.2.1 The bigger picture ........................................ 111
  6.2.2 Overview of SyMAD ................................... 112
  6.2.3 Case study: A simplified banking system .................. 112

7 Software architecture description ........................................ 127
  7.1 Kruchten 4+1 Views ....................................... 127
    7.1.1 Views .................................................. 128
    7.1.2 Criticisms raised at Kruchten 4+1 ...................... 129
  7.2 Architecture Description Languages (ADLs) ..................... 129
    7.2.1 Scope of an architectural description .................. 129
    7.2.2 Architecture description approaches .................. 130
    7.2.3 Aspect-Oriented ADL .................................. 130
  7.3 IEEE/IEC/ISO 42010 ...................................... 134
    7.3.1 History of IEC/ISO/IEEE 42010 ......................... 134
    7.3.2 What does IEC/ISO/IEEE 42010 provide? ............... 134
    7.3.3 Software architecture definition ...................... 134
    7.3.4 Stakeholders and stakeholder concerns ............... 135
    7.3.5 Envisaged uses of architectural descriptions .......... 135
    7.3.6 Normative elements .................................. 135
    7.3.7 View points and views ................................ 136
    7.3.8 IEC/ISO/IEEE 42010 metamodel ......................... 137
  7.4 Architecture template documents ............................ 140
    7.4.1 ......................................................... 140
    7.4.2 Software architecture documentation template ........ 142

8 Software architecture recovery ..................................... 147
  8.1 What is software architecture recovery? ..................... 147
  8.2 Why perform a software architecture recovery? ............ 147
  8.3 Challenges of software architecture recovery .............. 148
  8.4 How would you recover a software architecture? ........... 148
  8.5 Steps of architecture recovery ............................ 149
  8.6 Approaches to software architecture recovery ............. 149
  8.7 A Systematic Method for software Architecture Recovery (SyMAR) .............................. 149
    8.7.1 Overview of SyMAR .................................. 150
    8.7.2 Request traces ......................................... 150

9 Software architecture analysis .................................... 153
  9.1 Overview .................................................. 153
    9.1.1 When is a software architecture analysis done? ....... 153
    9.1.2 Basic approaches to software architecture analysis ... 153
    9.1.3 Architecture analysis methods ........................ 154
  9.2 The Architecture Trade-off Analysis Method (ATAM) ......... 154
    9.2.1 Benefits of ATAM .................................... 155
    9.2.2 Applicability of ATAM ................................ 155
    9.2.3 Overview of the ATAM process ......................... 155
    9.2.4 The role players in ATAM ............................. 155
    9.2.5 Outputs of ATAM ..................................... 156
    9.2.6 The ATAM process ..................................... 157
## The Model Driven Architecture (MDA)

### 10.1 Overview of Model-Driven Architecture

- **10.1.1 Architecture and Application Design within MDA**
- **10.1.2 Problems MDA aims to address**
- **10.1.3 MDA standards**
- **10.1.4 Model-to-model transformations**
- **10.1.5 MDA tools**

### 10.2 MDA tools

- **10.2.1 Eclipse modeling tools**
- **10.2.2 Overview of ecore**
- **10.2.3 Overview of XMI**
- **10.2.4 Overview of the OCL**
- **10.2.5 Overview of QVT-Operational (qvt-o)**
- **10.2.6 Overview of QVT-Relational (qvt-r)**
- **10.2.7 Overview of EMFText**
Chapter 1

Introduction to software architecture

1.1 Is architecture a common concept?

We talk of the architecture of a building, a city an organization a machine and even of the architecture of software system. But is it really a common concept, and, if so, what is that concept? In other words, can we find a definition of architecture which could be applied across buildings architecture, the architecture of a city or a country, the architecture of an organization and also the architecture of a software system?

1.2 The purpose of architecture

In this book it is proposed that the purpose of architecture is to provide the specification of an appropriate software infrastructure within which application functionality is specified, deployed and executed, such that

1. the required access and integration channels are available
   • for example, to humans, machines (trains, software systems, . . .)

2. the infrastructural concerns or responsibilities are addressed
   • for example, to transport people, to store information, to provide resources, . . .

3. the required quality attributes can be provided
   • for example, scalability, reliability, security, performance, integrability, monitorability, auditability, flexibility, portability, deployability, cost, . . .

From this perspective architecture addresses non-functional requirements whilst application design (e.g. system functionality, business functionality or domestic process design) addresses the functional requirements.
1.3 Questions addressed by software architecture

Below is a list of some questions which are commonly raised around software architecture:

1. What are the architectural components (infrastructural software components) and what are their responsibilities?
2. How do the architectural components communicate?
3. How will the system handle the number of concurrent users?
4. How will authentication, authorization, confidentiality be enforced?
5. How will we ensure reliability and fail-over safety?
6. Does the software architecture support pluggability, to what extend and how?
7. Is logging supported by the software architecture, and if so, how?
8. What frameworks, technologies, protocols, ..., will be used?
9. How does the software architecture support monitorability and auditability?
10. How does the system deployed or ported?
11. How is maintainability supported. Can the system support life maintenance, and if so, how?
12. How is high-performance achieved?
13. What frameworks and technologies are used and why?

1.4 What is software architecture?

1.4.1 Suggestions?

Before starting to introduce some definitions of software architecture, it would be good if you pondered first about the concept. How would you define software architecture and what would you say is the purpose or responsibility of software architecture?

1.4.2 Definitions of software architecture

We do not yet have a general consensus on what exactly software architecture is. A common joke is

If you get five architects together and ask each for a definition of software architecture, you’ll get seven different definitions. This is exemplified by the list of definitions for software architecture maintained by the Software Engineering Institute at Carnegie Mellon which lists over 300 different definitions. Note that we do not have such a problem in other domains like the definition of buildings architecture, quality assurance and so on.
1.4. WHAT IS SOFTWARE ARCHITECTURE?

1.4.2.1 Differences between software architecture definitions

The different definitions of software architecture largely share only one commonality — that an architectural description must specify components and connectors. However, there is no consensus on

- whether an architectural description specifies only high-level components or whether a software architecture needs to be specified across levels of granularity (and, if so, where the boundary is),
- how to distinguish between architectural and application responsibilities and hence between architectural and application components,
- whether the specification of functionality or processes addressing functional requirements is to be included in an architectural specification or not, and
- whether strategies/tactics are relevant to an architectural description (only the aspect-oriented ADLs have currently any support for this).

Different definitions of software architecture are supported by different architecture description languages (ADLs) which provide support for expressing the domain of intercourse required by that particular definition.

1.4.2.2 The choice of definition of software architecture has wide repercussions

The choice of definition for software architecture has very wide repercussions. It affects

- how the responsibilities and skills requirements for software architects, how a software architecture needs
- the contents of the architecture requirements specification,
- how we design an architecture,
- how we document it, and
- how we evaluate/validate it.

1.4.2.3 Classes of software architecture definitions

Scanning the many definitions of software architecture one can identify three major classes:

1. **Software Architecture is the high-level abstraction of software system.**
   This approach has been advocated by prominent practitioners and researchers like Martin Fowler and Manfred Nagl.

2. **The software architecture is the structure and externally visible properties of the software system.**
   - Perhaps definition which is currently most widely used.

3. **Fundamental concepts & constraints within which software system is designed & developed**
1.4.2.4 Software architecture as a high-level abstraction of a software system

This class of definitions raises a number of questions. In particular, where is the boundary of “high-level”? If one starts with a component of a system, is there still software architecture? How do we know?

Also, what then is exactly the difference between architecture and application design? Is it simply a matter of granularity? This class of definitions requires typically multiple views to specify a software architecture. Some views might focus on process specifications, other views might focus on structural aspects of the software system.

For example, the Kruchten 4+1 Views has been quite widely used to describe software architectures. The approach requires that one specifies the following views:

- **Logical View** which specifies the functionality provided to the end user using class, component and sequence diagrams.
- **Development View** which depicts the modularization of the system using package and component diagrams.
- **Process View** which is used to specify the system processes via activity diagrams.
- **Physical View** used to specify the physical deployment of system components and the physical connections between them using deployment diagrams.
- **Use Case Views** showing representative use cases and scenarios using use case and sequence diagrams.

Within such an approach architecture and technology neutral application design is typically not supported. It would be nice, however, to support the ability of deploying the same functionality into different software architectures addressing different non-functional requirements.

1.4.2.5 Software architecture as the structure and externally visible properties of a software system

The first version of the IEEE specification for the requirements of an architectural description, IEEE-1471, defined software architecture as follows:

**Definition 1.4.1.** Software architecture is the structure or structures of system which comprise software elements, their externally visible properties, and the relationships amongst them.

This definition does not focus on application functionality. Instead the main focus is on the structure, i.e. the components and the relationships between them, and the properties which are generally regarded the quality attributes of the system. But is...
1.4. WHAT IS SOFTWARE ARCHITECTURE?

architecture the properties or should it rather include the specification of the tactics which are used to realize those properties?

This class of definitions does not really allow for the specification of levels of granularity for the software architecture. There is also still no still no clear guideline on how to determine the boundary between architecture and application design, though such a separation is not explicitly excluded by the definition. Once again, how do we know whether a component is an architectural component or an application component?

The imaturity of the field and the uncertainty around the concept of software architecture are exemplified by the IEEE changing its definition of software architecture in its 2011 version of the IEEE specification for the requirements of an architectural description. ISO/IEC 42010:2011. The new definition reads now as follows:

**Definition 1.4.2** (ISO/IEC 42010:2011). Software architecture is the fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and the principles of its design and evolution.

This definition is a lot more abstract and does allow for the view fundamental concepts and constraints as the concepts and constraints within which application functionality is to be developed within that software architecture. For example, a services-oriented architecture introduces the concept of a service with constraints that services must be stateless, discoverable and published as WSDL contracts. The definition does not, however, refer to strategies or tactics used in the software architecture to address quality requirements.

1.4.3 Reference architectures

1.4.3.1 Reference architectures, frameworks and instance architectures

A reference architecture can be defined as follows:

**Definition 1.4.3.** A reference architecture is a domain-specific architectural template which aims to address architectural concerns for a particular class of problems.

A reference architecture specifies an infrastructure which addresses the common architectural concerns for that domain. Examples of reference architectures for different domains include Java-EE for enterprise systems, AUTOSAR for vehicular software, SOA for integrating many systems, and Space-Based for software requiring complex decision making.

For most reference architectures there is a choice of implementing frameworks. These frameworks provide a partial or complete implementation of a reference architecture. For example, JBoss Application Server and Apache Geronimo are two implementing frameworks of the Java-EE reference architecture. Similarly, Arctic Core and Continental Engineering Services provide implementations of the AUTOSAR reference architecture. Apache Axis and Open-ESB are SOA frameworks and Gigaspaces is a framework which implements the Space-Based reference architecture.

An architecture for a specific system is a defined architecture or an instance architecture. It may or may not be based on reference architecture and may or may not use a framework.
1.4.3.2 A reference architecture as pure architecture

One can argue that reference architectures are specifications of pure architecture. They do not contain any specification of application functionality. For example, Java-EE does not contain any aspects which is specific to banking, retail or insurance systems. Similarly AUTOSAR does not contain any elements for motor or brake control. However, we can deploy banking, retail or insurance applications into Java-EE and software systems monitoring and controlling brakes or engines into AUTOSAR. The reference architecture does not provide any application functionality, but it does provide an infrastructure addressing architectural concerns and within which the non-functional requirements can be realized.

One can thus argue that one can reverse engineer a reference architecture to obtain a definition of architecture itself and that we even have implementations of “pure architectures” — the implementing frameworks or reference architectures.

1.4.3.3 The Java-EE reference architecture

Figure 1.1 provides an overview of the Java-EE reference architecture and its core components. The component for the first level of granularity is the application server itself. Its core responsibilities include Component = Application server

1. providing access to users (human and system users),
2. providing an infrastructure for processing business logic, and
3. providing an infrastructure for integrating with backend service providers like persistence providers and systems providing lower level services.

These high level responsibilities are assigned to next level granularity components. In the case of Java-EE these are the web container, EJB (Enterprise Java Beans) container, JPA (Java Persistence API) persistence context.

The reference architecture is designed to address the quality requirements for typical enterprise systems. These include reliability, scalability, security, integrability and flexibility.

Structural patterns are used to constrain the infrastructure between components. Java-EE uses the layering pattern with the second level granularity components organized in layers (one layer can only access the next lower layer).

Java-EE also specifies a range of architectural tactics to address the quality requirements of typical enterprise systems. It specifies, at this first level of granularity tactics like clustering and interception to increase scalability, reliability and flexibility.

Note that the first level granularity component, the application server as a whole, does not host any application components. These are hosted by the lower level components like the EJB container and the Web container.

The second level of granularity components are the web and EJB containers and the JPA provider. The infrastructure of these is again based on architectural patterns, for example the Model-View-Controller (MVC), Controller and Layering patterns. Further architectural tactics or strategies are used to address the quality requirements for enterprise systems. These include caching, resource reuse and interception.
1.4. WHAT IS SOFTWARE ARCHITECTURE?

Figure 1.1: Overview of the Java-EE reference architecture.

But these lower level architectural components do host application components. For example, web containers host facellets, backing beans and binding beans. Similarly the EJB container does host stateful and stateless session beans and the persistence context hosts entities.

1.4.3.4 Services Oriented Architectures

Figure 1.2 provides an overview of the Services Oriented reference Architecture (SOA).

1.4.3.4.1 2nd level of granularity One of the second level granularity components is the process execution engine. Focusing on that component for this discussion, we not that it once again implements a number of architectural tactics or strategies to address quality requirements. These typically include clustering, thread pooling, interception.

The process execution engine does host application components. In SOA application components are specified using stateless services. Higher level services are “orches-
trated” across lower level services using the pipes and filters pattern, i.e. the infrastructure between application components is thus constrained by this architectural pattern.

A number of constraints apply to the application components specified by this reference architecture. This includes that services may not maintain state across service requests, that they must be self-healing (a failure in processing one request should not affect the way in which subsequent requests are processed), that they must be discoverable, that they must be pluggable components by virtue of realizing published contracts, and so on.

1.4.3.5 Insights obtained from analyzing reference architectures

Analyzing these reference architectures which we regard as “pure” architectures we come to the following conclusions:

- In many software architectures we are able to distinguish between architectural and application components which respectively address application and infrastructural concerns with application components addressing functional requirements for application users and architectural components focusing on non-functional requirements.
- There is a need to specify architecture across levels of granularity.
- At any level of granularity there are architectural responsibilities which are assigned to architectural components.
- The infrastructure at any level of granularity may be constrained by structural patterns.
- At any level of granularity the architecture may use strategies or tactics to address quality requirements of the software system.
- For those architectural components which host application components, the software architecture may specify concepts and constraints for the application components hosted within the architecture. Note that even though the architecture
introduces concepts and constraints for application components, it does not actually provide any application components.

1.4.4 Definition of software architecture

We define software architecture as follows: Software architecture is the specification of the software infrastructure within which application logic providing user functionality can be specified, deployed and executed.

In this context application functionality will address the functional requirements of application users whilst the architecture is meant to address the non-functional requirements.

1.4.5 Components of software architecture

A software architecture specification should specify across levels of granularity

1. A set of architectural responsibilities and the architectural components to which these are assigned. Common elements of architectural responsibilities include those of
   • providing access and integration channels,
   • providing a computational environment,
   • providing a persistence infrastructure

   Specific problem domains typically include further architectural responsibilities.

2. The architectural components are organized within an infrastructure which may be constrained by architectural patterns specifying the integration channels between them.

3. The architectural strategies used to concretely address quality requirements.

4. The concepts & constraints within which application logic is to be specified.

1.4.6 Typical responsibilities of architecture

The purpose of software architecture is to

• provide a suitable infrastructure for application components providing user functionality
• which addresses the non-functional requirements.

Typical examples of responsibilities include

• providing access and integration channels to humans and systems,
• managing resources effectively to achieve cost-effective scalability,
• enforcing security including confidentiality, authentication, authorization and non-repudiation,
• provide fail-over safety across all components to ensure a level of reliability,
• providing an infrastructure for flexible, maintainable, application development, and
• providing an infrastructure for reliable, efficient and auditable process execution.
CHAPTER 1. INTRODUCTION TO SOFTWARE ARCHITECTURE

1.4.7 Pure application code

Application developers should be able to focus on developing application functionality, leaving technical concerns to software architecture. This includes the development of so-called “plumbing code” like the code which maps objects onto a persistence infrastructure like a relational database, en/decoding, request routing. Also responsibilities like thread pooling and load balancing, object caching and authorization should not have to be addressed in application code as these concern themselves around non-functional requirements. Application functionality should be devoid of “plumbing code”.

1.4.8 Architecture neutral design and implementation

There is a general movement towards model-driven development and architecture and technology neutral analysis and design using methods like the Use-Case, Responsibility Driven Analysis and Design (URDAD) method. In such a model-driven approach the modeling is done in the problem domain and the mapping onto different architectures and technologies can be partially or completely automated.

Furthermore, there is also a trend in modern software development to develop code in an architecture independent way. This is commonly done using metadata to specify any information which is specific to particular software architectures. For example, code artifacts annotated with both, Java-EE and Spring annotations can be deployed in either of these architectures.

1.4.9 Interplay between architecture and application design

The architecture does not address any of the functional requirements of the user (application). It provides an infrastructure within which functionality is developed, deployed and executed. But, to implement architectural tactics or strategies like load balancing we do require functionality. This is, however, not application functionality, but functionality provided by the software architecture – the functionality does not address functional requirements of the user.

Note that one can use URDAD to design such architectural functionality.

1.4.10 Architecture as a matter of perspective

Note that whether a system component is an architectural or an application component is a matter of perspective. If we, say, design a banking system then the application servers and naming and directory services are architectural components which have been chosen to provide a suitable infrastructure for our application components addressing the non-functional requirements.

On the other hand, if your system is the application server itself, then that system has both application design which provides user functionality to the users of the application server and architectural design to address the non-functional requirements of the application server itself. The user in this case might be the developer developing a banking system and the functional requirements include now aspects like mapping an object onto some persistence technology, that of performing authorization against
the user roles of an authenticated principle and that of providing a handle to the
transaction manager. From the perspective of the banking system these concerns
were architectural concerns, but from the perspective of developing an application
server, these address functional requirements of the application server.

There are also architectural requirements which need to be addressed. Examples may
be portability, providing access to system resources like threads or memory and so
on. These would be addressed by the architecture of the application server which
may include a bridge to the underlying operating system (e.g. the Java Runtime
Environment, JRE) which provides access to resources obtained from the operating
system and ensures portability across operating systems.

## 1.5 The bigger picture

Currently in model-driven development we define transformations which can map an
application design onto a specific software architecture, i.e. we are currently hard-
coding the software architecture into the transformation elements. In future we would
like to be able to generate code implementing application functionality as specified
in application design for architecture as specified in architecture model. This vision
is shown in figure [1.3](image)

## 1.6 Wish list

In order to be able to generate an implementation providing the application func-
tionality as specified in an application design model within a software architecture as
specified in a software architecture model we require
• a formalized architecture analysis and design method, i.e. a method which is similar to URDAD, but for architecture design, and
• Architecture Description Language (ADL) which supports the semantics required to describe the software architecture.

The ADL will require support for specifying components and connectors, infrastructure constraints in the form of structural patterns, tactics or strategies used to address quality requirements. In addition, the language needs to be able to specify concepts and constraints within which application components for that software architecture are to be developed.

Ultimately we also need good tool support.

### 1.7 Software architecture in the context of MDD

![Software architecture in the context of MDD](image)

Figure 1.4: Software architecture in the context of model-driven development.

Figure 1.4 depicts software architecture in the context of a model-driven process.

### 1.8 Responsibilities of software architecture

The responsibilities of a software architect include the following:
1.8. RESPONSIBILITIES OF SOFTWARE ARCHITECTURE

- Assisting clients to specify architecture requirements.
- Design and document a software architecture.
- Validate existing and proposed software architectures.
- Assist with non-functional testing.
- Recover software architectures.
- Evolve architectures / re-architect.
- Ensure software architecture compliance.
- Assist developers to understand the rationale behind the software architecture as well as the technicalities around developing the software architecture. This includes identifying training needs within the development team.
- Liase with client to ensure software architecture aligned with business architecture.
Chapter 2

Structural patterns

2.1 What are structural patterns

The specification of structural patterns is one of the central architectural decisions which typically has wide- and long-ranging consequences. We define a structural pattern as follows:

**Definition 2.1.1.** A structural pattern is a template solution for a structure which has been shown to be able to address specific architectural concerns.

Structural patterns specify architectural components or abstractions of architectural components and optionally the responsibilities each component needs to address. Additionally structural patterns specify connectivity constraints between these architectural components, i.e. the infrastructure between the architectural components.

2.2 Why use structural patterns?

Each pattern is aligned with certain qualities, i.e. using a pattern makes it easier to achieve certain quality attributes usually at the cost of other quality attributes. Like many architectural decisions, the selection of a structural pattern implies a trade-off between quality attributes.

Specifying or identifying the structural patterns used within a software architecture improves the understanding of a software architecture, simplifies the process of assessing a software architecture and comparing alternative architectural solutions and also facilitates communicating aspects of a software architecture effectively.

Structural patterns are also useful in the context of having to understand and assess vendor solutions (e.g. vendor frameworks).

2.3 Layering

In the case of the layering pattern, components organized in layers, each of which is ultimately assigned some responsibility. The responsibilities for the individual layers
is not prescribed by the pattern but is allocated when the pattern is used within a design.

![Layered Pattern Components](image)

Figure 2.1: In the layered pattern components in one layer may only access components which are either in the same or the next lower layer.

The structure prescribed by the layering pattern is shown in figure 2.1. The infrastructural constraint introduced by the layering pattern is that components in one layer can only access components which are either in the same layer, or in the next lower level layer.

### 2.3.1 Uses of the layering pattern

The layering pattern is used widely. Many organizations are based on a layered architectural pattern. For example, some organizations have the following layers:

1. Client access layer (dealer, web pages, call center, web services layer, ...)
2. Front-office layer (providing client-faced services)
3. Back-office layer (providing back-office services)
4. Infrastructure layer (integration with suppliers, regulatory institutions, ...)

Also, many enterprise systems are traditionally based on the layered architectural pattern starting with the early client-server based systems through to 3, 4 and 5 tier architectures.

Even your sound system is typically based on the layered architectural pattern with the layers being typically the following:

1. signal source layer (media stream player, radio, CD player, ...)
2. signal modification layer (pre-amplifier)
3. amplification layer (power amplifier)
4. rendering layer (speakers)
Another common example of the use of layering is that of network protocols. For example, the TCP/IP protocol has the following layers:

1. Application layer (e.g. HTTP)
2. Transport layer (TCP)
3. Internet Layer (IP)
4. Network access layer (Ethernet, 80211.n, PPP, . . . )

### 2.3.2 Advantages and disadvantages of layering

The benefits of using layering include

- having *pluggable layers* and being able to replace single layers (e.g. the presentation layer or a backend system),
- improved *cohesion* through defined higher-level responsibility localization,
- *complexity reduction*, particularly for larger systems,
- *loose high-level coupling*
- improved *testability* through the ability to mock out lower layers,
- improved *reuse* of high-level components, and
- improved maintainability through the ability to have different layers developed independently by different teams with different skills sets.

The challenges of Layering include

- *inflexible* due to rigid structure and communication constraints,
- *performance overheads* caused by communication across layers which may require encoding/decoding across layers,
- *high maintenance costs* with changes to lower layers impacting on higher layers, and
- difficult to encourage *innovation* because people or components see just a small world other areas are controlled by other hierarchies.

### 2.4 Microkernel

The core of a microkernel based infrastructure is an integration bus which is responsible for message or request routing. Internal servers are meant to provide backbone or basic infrastructural services and are typically required to be robust, reliable and slowly evolving. External servers typically provide higher-level client facing services and are generally required to be more flexible. An adapter provides a clients a single access point to services offered by external servers.
2.4.1 Uses of the microkernel pattern

Example applications of the microkernel pattern include Services Oriented Architectures (SOAs) with the microkernel being the Enterprise Services Bus (ESB). Operating systems are commonly based on the microkernel architectural pattern with the internal servers being providing access to and managing system resources like memory, threads, processes, network connections and so on. These are required to be very reliable and predictable and typically do not change frequently.

Many organizations are based on the microkernel pattern. Such organizations are commonly called Services Oriented Enterprises (SOEs). The internal servers are slowly evolving internal (back-office) services like transaction processing, regulatory reporting, and procurement. The external servers are the client faced services which generally need to be flexible and evolve often rapidly.

2.4.2 Benefits and challenges of the microkernel pattern

The microkernel pattern provides an infrastructure for flexible client faced services on robust slowly evolving core/internal services. Benefits of the microkernel pattern include

- simplified integration with each component only needing to be able integrate with the integration bus,
- improved flexibility through simple pluggability and the ability to have different deployment configurations,
- simpler portability because you only need to plug in a different concrete lower-level services layer,
• improved reuse through simpler integration and because the adapters make component accessible independent of differences in implementation technologies, interfaces and communication protocols, and
• improved maintainability due to the separation of low-level services from high-level services.

Concerns around the microkernel pattern include

• Performance concerns due to communication and routing overheads,
• Reliability concerns in the case where the integration bus is single point of failure, and
• that the infrastructure makes process management more complex if the latter is not integrated into the architecture.

2.5 Blackboard

Assume you have a difficult mathematical problem to solve and that you have asked each of a group of mathematicians whether they could solve it for you, but none managed to solve it. You also do not have a process to get you to the solution. Such a process might have involved asking one of the mathematicians whom you know can solve sub-problem 1 followed by another who you know can solve sub-problem 2 and so on.

So, how else can you get your problem solved? You could ask the group of mathematicians to solve the problem together. You will give them a blackboard (this gives away the age of the pattern) and let them try and auto-orchestrate a process which solves the problem — you could use the blackboard pattern.

![Figure 2.3: The structure of the blackboard pattern.](image)

Figure 2.3 shows the structure of the blackboard pattern, depicting the communication links between the controller, the expert pool and the blackboard itself.

The elements of the blackboard pattern are the following:

1. A blackboard or knowledge repository which hosts the problem specification as well as the current state of solution,
2. A pool of experts or processing units which decide themselves when to contribute towards the process of solving the problem and which also perform quality assurance on any contributions to or modifications of the blackboard.
3. A **controller** who is minimally involved in solving the problem itself, but who specifies the problem or feeds problems into the blackboard, feeds experts into the expert pool and is involved in conflict resolution between experts.

### 2.5.1 Uses of the blackboard pattern

The blackboard pattern is widely used where you need to collaboratively perform a task without having a defined process for that task. Often the problem requires a level of innovation.

For example, *software development, collaborative design* and *knowledge capturing* are commonly done within blackboard based architectures – often within an open source approach, i.e. group of open-source developers, society as a whole, ... In such projects the blackboard is, for example, the version control system or wiki, the expert pool is the set of experts (e.g. a software development or marketing team), and the controller is, for example, the project lead. Quality assurance is done by the group of experts themselves – i.e. they generally quality assure any contributions done by any of the other experts. The project lead specifies the scope of the problem as well as the general vision for the solution.

Blackboard based architectures are also widely used for monitoring and control systems. Information (e.g. events) is fed into the blackboard where it is observed by monitoring and control units. Monitoring units render observed events to users. Complex event processors may generate higher level events/data from observing certain data patterns, generating new data/events which is put back onto the blackboard where they may be observed by other monitoring and control units. Control units react to events by performing certain control activities. Examples include monitoring and control systems for industrial plants, cockpits, organizations and so on.

The *Space Based Architecture* (SBA) is a reference architecture which is based on the blackboard architectural pattern. It is commonly used for application domains requiring complex decision making. Examples include automated trading, airport traffic control, military systems which detect and track objects, and so on. Frameworks based on the blackboard pattern include *JavaSpaces*, *GigaSpaces*, and *Apache River*.

The blackboard pattern is widely used outside software architecture where teams need to come up with innovative solutions. For example, research and marketing teams often operate in the context of blackboard based architectures. Also JAD sessions are based on the blackboard pattern, ... , ...

### 2.5.2 Benefits and concerns around the blackboard pattern

The benefits of the blackboard pattern include

- that the pattern provides an infrastructure for generating *innovative solutions* and an ability to solve difficult problems,
- that it is easy to achieve *scalability* across a grid of processors subject to having scalable implementation of the blackboard itself,
- that the pattern facilitates continuous *quality* optimization, and
• that the pattern provides a high level of flexibility and maintainability — adding experts who are able to perform new processing steps may immediately change a range of auto-orchestrated processes.

2.6 Master-Slave

In the Master-Slave pattern is used in environments where large-scale concurrent processing is possible and required. A master distributes work across a processing grid of slaves with the slaves doing the main work and the master managing the overall process. The high-level structure is illustrated in figure 2.4.

Distributing the work may require transporting the data for each slave to the slave. Alternatively the slave can get local access to data within a distributed persistence infrastructure like a distributed file system or distributed database.

The Master-Slave pattern often requires that one uses aggregators or other reduces to calculate some statistical or inferred result from the large amount of data processed.

In order to achieve a level of monitorability and reliability, one generally implements

• job and task tracking, as well as
• task restart on failure.

Figure 2.4: The master-slave pattern a master distributes work across a typically large number of slaves often located on different nodes.
2.6.1 Uses of the master-slave pattern

The Master-Slave is commonly used when one needs to do similar type of processing across a large bulk of data. It forms the basis for the Map-Reduce reference architecture and hence also for implementing frameworks like Hadoop Map-Reduce.

It is widely used for batch processing, data acquisition and areas requiring processing of large amounts of data like search engines, statistical analysis of stock data, vote counting and so on.

2.6.2 Benefits and concerns

The main benefits of the Master-Slave pattern is the ability to achieve a high level of scalability through splitting the work into independent sub-tasks executed by independent nodes on a slave grid.

Reliability and fault tolerance as well as monitorability/auditability is usually achieved through job and task tracking. The master will resubmit work which has not been successfully completed by one of the slaves.

Concerns which may prevent you from using the Master-Slave pattern include that it may be difficult to divide labour and data, that it is not suited for interactive processes and that it is difficult to guarantee in the context of distributed processing. The latter is particularly relevant when processing data for different clients within the same infrastructure (e.g. a cloud-computing infrastructure).

2.7 Hierarchical

The structure of the hierarchical pattern is illustrated in figure 2.5. Hierarchical is based on a recursive containment hierarchy similar to the composite pattern. It is also used in the context of inheritance hierarchies.

![Hierarchical Structure Diagram](image)

Figure 2.5: In a hierarchical structure a node can only communicate with its parent node and its child nodes and otherwise through defined communication channels.

The pattern introduces the infrastructural constraint that nodes may only communicate with parent and child nodes and otherwise through defined communication channels.
2.7. HIERARCHICAL

2.7.1 Uses of the hierarchical pattern

The hierarchical pattern is used across a wide range of domains. For example, organizations are commonly based on the hierarchical pattern with (authoritarian) reporting and accountability hierarchies.

The basic structure of the master-slave pattern could be seen as a single layer hierarchy, but one can also have hierarchical master-slave based systems where masters across levels in the hierarchy distribute work amongst their slaves.

The hierarchical pattern is also used for inheritance hierarchies where commonalities including common requirements (interfaces) and processes (method bodies) are enforced through inheritance. It is also commonly used to provide a base infrastructure for auditability and other non-functional concerns.

Hierarchical databases are widely used for the persistence infrastructure supporting naming and directory services. The pattern facilitates inheritance of certain authorization related parameters, i.e. that child nodes inherit aspects of parent nodes with the option of overriding these.

The structure of documents is based on the hierarchical pattern with higher-level sections containing lower level sections documents (sections have sections). Such a pattern potentially allows section reuse at different hierarchical levels.

Another example is that of styling frameworks like that of Cascading Style Sheets (CSS). By default a child node inherits the styling of the parent node, but may override that styling.

2.7.2 Benefits and challenges of the hierarchical pattern

Some of the benefits of using the hierarchical pattern include

- the ability to achieve a high-level of performance and scalability (e.g. through easily finding things in a hierarchical database and through work distribution in a hierarchical master-slave system),
- improved reliability through oversight/accountability hierarchy,
- responsibility localization across levels of granularity, and
- improved maintainability through inheritance.

Concerns around the use of the hierarchical pattern include

- reduced flexibility due to the rigid structure enforced by the pattern,
- reduced ingenuity due to components or people only getting exposure/access to only a small subset of world and changes not easily implementable because other areas are under the control of another hierarchy, and
- increased cost due to overheads of higher-level, non-processing layers.

Note that in an organization where promotion is not carefully managed the hierarchical pattern may lead to individuals being promoted until they reach their personal level of incompetence.
2.8 Pipes and filters

2.8.1 The pipes & filters pattern

In the pipes and filters pattern, filters are connected by pipes. The filters are the processing units and the pipes the communication channels.

In the pipes and filters pattern, the filters are commonly either pure functions or stateless services, i.e. state may be maintained whilst processing a request, but does not survive through to next request. A filter does not depend on a preceding or following filter, i.e. filters are independent of one another. The output of functions depends solely on the inputs as pure functions cannot access any information from the environment, nor can they alter the state of the environment.

Each filter has an input and an output pipe. Additionally there may be other pipes like error or logging pipes.

Higher level pipes (stateless services or functions) are assembled from lower level pipes.

Example uses of pipes and filters include

- **Unix command processing**

  ```
  ls | head -4 | tail -2 > fileContaining3rdAnd4thFileNames.txt
  ```

- **Functional programming**

- **Media streaming/processing pipelines** which perform a sequence of operations on media streams including encoding and/or decoding operations, object detection, image or sound manipulation and so on.
  
  - decoding, object detection, image/sound manipulation, ...

- **I/O streaming**

  ```
  objectIS = new ObjectIS(new HashingIS(new BufferedIS(socket.getIS())));
  ```

- Service Oriented Architectures (SOAs) orchestrate higher level services from lower level services connected by queues.

- In the manufacturing sector one finds filters in the form of machines connected by pipes in the form of conveyor belts.

- Workflow systems which may include filters comprising of manual workflow steps where the processing is done by humans.

The pipes and filters pattern is widely used due to the flexibility it provides. The main benefits of the pattern include

- **Flexibility** and **Time-To-Market** as one can easily modify a workflow by adding, removing, or replacing filters and one can easily assemble new workflows from the available functionality (available filters).

- **Reuse** and **Testability** due to the decoupling of processing units and even more so if pure functions are used for the pipes.
2.9. MODEL-VIEW-CONTROLLER (MVC)

There are, however, also a number of concerns one needs to keep in mind when using this pattern. These include

- potential performance issues caused by communication overheads, en- and decoding,
- that it is difficult to achieve ingenuity within a pipes and filters processing environment as processing units only see their inputs and outputs,
- that it might be non-trivial to achieve a high level of reliability as there is no high-level control or oversight, and
- that it is difficult to implement interactive processes.

2.9 Model-View-Controller (MVC)

Most presentation layer architectures are based on the Model-View-Controller (MVC) pattern. The aim of the pattern is to reduce presentation layers complexity by separating the following three responsibilities:

1. Provide view onto information \( \rightarrow \) View
2. React to user events \( \rightarrow \) Controller
3. Provide business services & data \( \rightarrow \) Model

Aims of the MVC pattern include that the three components can evolve independently and that the components can be deployed onto different nodes.

Figure 2.6 depicts the structure of the MVC pattern. Note that the MVC pattern fully decouples the model from both, the view and the controller. The model is usually observed by all views, facilitating that the views can update themselves on model changes. Additionally, the model may also be observed by the controller to apply control logic changes on model change events.

Figure 2.7 depicts the dynamics of the MVC pattern, showing how updating the model through one view results in other views updating themselves.
2.9.1 Uses of the MVC pattern

Most user interface libraries like Smalltalk, JavaFX, QT, Swing (sort-of) are based on the MVC pattern. In addition, most modern web-based presentation layer frameworks like JSF, Django web framework, Spring web framework, Struts, ASP.Net, ... are based on the MVC pattern.

2.9.2 Benefits and concerns around the MVC pattern

Benefits of the MVC pattern include

- a general simplification through separation of concerns,
- improved reuse, particularly of the model and view components,
- improved maintainability, facilitating that the model, view and controller can be developed by different teams (back-end developers, UI designers and front-end developers),
- improved testability, particularly around being able to test the model independent of the UI, and being able to test the UI with a mock model

However, there are also a number of concerns around the MVC pattern. In particular,

- maintainability can be compromised because model changes may impact the view and the controller,
- more complex message patterns may result in reduced performance, particularly if the components are not co-located if messages need to be encoded and decoded, and limited reuse due to coupling of the view and the controller to the model

Some of the above are mitigated by introducing an adapter or a bridge between the UI (view and controller) and the model.
2.10 Patterns and quality attributes

<table>
<thead>
<tr>
<th>Tactic</th>
<th>Performance</th>
<th>Scalability</th>
<th>Reliability</th>
<th>Security</th>
<th>Availability</th>
<th>Integrity</th>
<th>Flexibility</th>
<th>Reusability</th>
<th>Maintainability</th>
<th>Testability</th>
<th>Affordability</th>
<th>Usability</th>
<th>Deployability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layering</td>
<td>▽</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>Pipes &amp; filters</td>
<td>▽</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>Microkernel</td>
<td>▼</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>Blackboard</td>
<td>▼ ▲</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>Comp Graph</td>
<td>▽ △</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>Bridge</td>
<td>▽ △</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>Hierarchical</td>
<td>△ △</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>MVC</td>
<td>▽ △</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>Master-Slave</td>
<td>△ △</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
</tbody>
</table>

Table 2.1: Impact of architectural patterns on quality attributes.

Table 2.1 depicts how the selection of structural patterns can positively or negatively impact on quality requirements. This table is useful to check which patterns are better aligned with the quality requirements and which other quality attributes could be negatively impacted or enhanced by selecting a particular structural pattern.

2.11 Exercises

1. For each structural pattern, try and find one or two examples in systems you have been working on. Discuss
   a) how the pattern is manifested in that system,
   b) why the pattern was used for that system,
   c) the architectural trade-offs being made by using that pattern,
   d) whether you agree with the use of the pattern or whether you would suggest an alternative pattern instead.
Chapter 3

Architectural tactics

Structure patterns do not address quality requirements. They provide an infrastructure within which quality requirements can be realized, i.e. an infrastructure which is aligned with a quality requirement.

For example, the pipes and filters pattern improves responsibility localization, flexibility and reuse whilst the blackboard pattern enables one to achieve scalability and reliability, innovation and the ability to solve difficult problems without having a defined process for doing this.

Structural pattern thus do not concretely realize quality requirements. We need to do certain things to concretely address the quality requirements for the system.

3.1 What is an architectural tactic?

Architectural tactics are also known as architectural strategies. They were introduced by Bass, Clements aal tactics in their book, Software Engineering in Practice, end edition [?].

Bass, Clements and Kazman define an architectural tactic as follows:

Definition 3.1.1. An architectural tactic is a design decision that influences the control of a quality attribute response.

Architectural tactics represent more abstract design decisions which are often addressed by more concrete design patterns which implement these architectural tactics. The aim of these tactics is to concretely realize desired quality attributes.

3.2 Why use architectural tactics?

Architectural tactics enable one to specify conceptually how you are aiming to address quality requirements. They can assist you with

- identifying ways to address quality requirements,
- making trade-off decisions more explicitly,
• understanding vendor products, frameworks or implemented architectures,
• deciding which features to use from reference architectures & frameworks, and
• communicating architectural decisions.

### 3.3 Some examples of architectural decisions

To make the concept of an architectural tactic more concrete, consider the following examples of architectural tactics:

- **Thread pooling** to improve scalability.
- **Clustering** to improve scalability and reliability.
- **Interception** for security and auditability.
- **Run-time lookups** for pluggability/flexibility.
- **Queuing** for reliability and scalability.

### 3.4 Scalability and Performance

![Figure 3.1: Goals for scalability and performance and some tactics commonly used to concretely address these quality requirements.](image-url)
3.5 Reliability

Figure 3.2: Goals for reliability and some tactics commonly used to concretely address this quality requirements.
3.6 Security

Figure 3.3: Goals for security and some tactics commonly used to concretely address this quality requirements.
3.7 Modifiability and extensibility

Figure 3.4: Goals for modifiability and extensibility and some tactics commonly used to concretely address these quality requirements.
3.8 Accessibility and integrability

Figure 3.5: Goals for accessibility and integrability and some tactics commonly used to concretely address these quality requirements.
### 3.9 Tactics and quality attributes

<table>
<thead>
<tr>
<th>Goal</th>
<th>Tactic</th>
<th>Quality Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Performance</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Resource reuse</td>
<td>Thread pooling</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Caching</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Connection pooling</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Virtualization</td>
<td>△</td>
</tr>
<tr>
<td>Spread load across time</td>
<td>Queuing</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Scheduling</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Preprocessing</td>
<td>△</td>
</tr>
<tr>
<td>Spread load across resources</td>
<td>Grids</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Clusters</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Load balancing</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Paging</td>
<td>△</td>
</tr>
<tr>
<td>Reduce communications load</td>
<td>Compression</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Bathing</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Course grained services</td>
<td>△</td>
</tr>
<tr>
<td>Optimization</td>
<td>Dynamic code optimization</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Dynamic query optimization</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Database indexing</td>
<td>△</td>
</tr>
<tr>
<td>Fault detection</td>
<td>Resource monitoring</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Checkpoint evaluation</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Deadlock detection</td>
<td>△</td>
</tr>
<tr>
<td>Fault prevention</td>
<td>Active redundancy</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Removal of faulty components</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Persistent messaging</td>
<td>△</td>
</tr>
<tr>
<td>Fault recovery</td>
<td>Passive redundancy</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Checkpoint rollback</td>
<td>△</td>
</tr>
<tr>
<td>Extensibility tactics</td>
<td>Interception points/layers</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Support aspects</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Contracts based decoupling</td>
<td>△</td>
</tr>
<tr>
<td>Service/ component publication</td>
<td>Naming service</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Trader service</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Interface/contract repository</td>
<td>△</td>
</tr>
<tr>
<td>Security tactics</td>
<td>Encryption</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Message integrity</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Authorization</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Authentication</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Request/connection dropping</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Minimize accessibility</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>Signature scanning</td>
<td>△</td>
</tr>
</tbody>
</table>

Table 3.1: Impact of different tactics on quality attributes. A solid upwards/downwards pointing triangle represents a strong positive/negative impact on a quality attribute and an empty triangle represents a moderate effect on the quality attribute. If a tactic has no symbol assigned for a quality attribute, that tactic has no significant effect on that quality attribute.
3.10 Exercises

1. Select an example system and identify the most important quality requirements for that system. For each of these quality requirements, in order of importance, select the architectural strategies which you suggest should be employed to realize the quality requirement.

2. Consider auditability as a quality requirement and
   • identify core auditability goals, and
   • identify, for each goal, a set of strategies you can use to concretely realize the architectural goal.
Chapter 4

Integration architecture

The integration architecture is one of the main elements of an architectural description. The decisions made around specifying the integration architecture commonly has a massive impact on a systems ability to address its quality requirements. It is particularly performance, reliability, security and scalability which are often strongly affected by the choices made when designing the integration architecture.

4.1 Integration challenges

Modern systems are commonly integrated into their environment. They often need to be accessible through a range of access channels through which humans and other systems can access their functionality and they often need to integrate with humans and systems to request services from them or provide information to them.

Core integration challenges can be categorized into

1. Accessibility challenges
2. Infrastructural constraints
3. Quality requirement challenges
4. Integration complexity
5. High-level work-flows

4.1.1 Accessibility challenges

A system which is to be integrated may not support certain integration channels, use proprietary technologies (e.g. protocols), provide incompatible APIs, or use incompatible data structures. All of the above make integration more challenging.

4.1.2 Infrastructural constraints

Infrastructure challenges may themselves complicate integration. In particular, it may be that there are no electronic integration channels, that the integration channels
use non-standard or proprietary technologies, or that they are simply of bad quality, i.e. not able to support the required throughput, or reliability.

4.1.3 Providing quality attributes across integration channels

The overall system qualities are affected by the quality attributes of integration. Typical quality requirements for integration include performance, scalability, reliability, security, auditability, flexibility, affordability and usability.

4.1.4 Lack of contracts

Without contracts it is not clear what exactly a component or service will do for you or, if you need to implement a service, what exactly is expected of you. There are also no guarantees of what is provided no guarantees that it is still provided tomorrow. If there is a problem, it will also not be clear on which side of the integration interface the problem lies.

Contracts are important for pluggability. If you develop against a contract, you will not have to change your implementation — at most you will have to add a new adaptor for the new component.

When specifying component or services contracts, the normal elements should be included, i.e. one needs to specify the data structures for the inputs and outputs of services or methods, the pre- and post-conditions and the quality requirements for the service provider.

4.1.5 Integration complexity

The integration complexity may be a significant part of the overall system complexity. In the case where many systems need to directly integrate with one another, there is a combinatorics problem. In the worst case there are \( \frac{n(n-1)}{2} \) integration surfaces. Each of these may use different protocols and may require its own infrastructure for data structure transformations, marshalling and de-marshalling.

Furthermore, the integration solution may be heavily impacted by changes and requires change control.

4.2 Integration concepts

Integration concepts include publication mechanisms, protocols, integration approaches, and integration mechanisms.

4.2.1 Component or service publication

Publication involves making services or components discoverable and publishing the information required to be able to use them. Generally one publishes the location of the service or component, the services offered, the required message exchange patterns including the data structures which are exchanged, the supported access channels, and the supported protocols.
4.2.2 Communication protocols

A communication protocol is defined as follows:

**Definition 4.2.1.** A *communication protocol* is the specification of the rules governing the syntax, the semantics, and the synchronization of communication.

A protocol may specify

- how a communication end point is found,
- how a communication session is established,
- how connection characteristics are negotiated e.g. whether one uses compressed communication or not, the encryption mechanism to be used, the character set used, any error detection or error correction codes which are used,...),
- the language/message structures used when communicating,
- how to start and end a message,
- how to detect whether the communication link is still active or not,
- what to do with corrupted or improperly formated messages (error correction, resend requests, ...), and
- how to terminate the communication session.

4.2.3 Integration approaches

There are four commonly used integration approaches:

- Shared resource
- Transferred resource
- Request-based integration
- Document-based integration

4.2.3.1 Shared-resource based integration

In the case of shared-resource based integration the state of a resource is modifiable by at least one role player and observable by at least one other role player. Examples includes shared database, file or object.

In the case of shared-resource based integration, the resource typically requires state protection of concurrent access. This is typically achieved through locking resources (e.g. object or record locks). Locks are of different strengths ranging from serialized access (very strong, to concurrent reads and exclusive writes, concurrent committed reads and single writes to concurrent uncommitted reads and single writes.

Locks often introduce the concept of a transaction Key to a transaction are the ACID criteria for transactions. They are

- **Atomicity:** done as a whole or not at all
- **Consistency:** if state consistency rules held prior to operation, will also hold after.
- **Isolation:** while one thread is busy with a resource, no other can access resource
- **Durable:** what’s done, cannot be tracelessly undone.
4.2.3.2 Transferred-resource based integration

In the case of transferred-resource based integration a resource is passed with request parameters or as a message. The resource may be specific for a specific service request as it is commonly in controller-based integration or it may be passed along from processing step to processing step as it is commonly done in pipes and filters based architectures like the messages in a Services-Oriented Architecture (SOA) or in the case of a production line as they are found in manufacturing or in software systems like a compiler constructing code.

The exchanged resource must be accessible to both sides. For this reason one either uses a standard encoding like XML data structures complying to the data structure specification of some schema which is accessible to all integration partners or one uses transformation elements which transform the resource from a form which is accessible to the one party to the form which is accessible to another party.

4.2.3.2.1 Document-based integration

In the case of document-based integration, the sender or source provides the information without specifying what should be done with the information. The processor or recipient receives the information and decides how to process it.

Document-based integration has a number of advantages over request-based integration including that the processor might be in a better position to decide what should be done with the information, that document-based integration supports the notion of multiple operations done on the same information. The approach is thus more flexible and the source and processing are less strongly coupled.

4.2.4 Integration mechanisms

4.2.4.1 Integration mechanisms

Examples of integration mechanisms include

- database-based integration,
- messaging,
- service-request based integration,
- space-based integration, and
- user-interface based integration.

4.3 Integration patterns

Enterprise integration patterns have been popularized by Gregor Hohpe and Bobby Woolf in their book “Enterprise Integration Patterns” [7]. Integration patterns as follows:

Definition 4.3.1. Integration patterns are proven, reusable, *generic integration solution components*, i.e. they represent specific *best practices* solutions to integration problems.
4.3. INTEGRATION PATTERNS

Like all patterns, integration patterns are seldom used in isolation, but instead commonly used in combination. The aim of any particular pattern is to address a particular concern, not a combination of concerns.

Ultimately a collection of integration patterns provides a dictionary of integration concepts.

### 4.3.1 Messaging benefits

Messaging has a number of benefits over more direct integration:

- **Decoupling** is improved as producer does not know who consumes the messages and both, the consumer or producer may change. Furthermore, there may be multiple producers and consumers.
- Messaging provides a simple infrastructure for **load balancing**. One simply adds more consumers processing off a queue.
- Messaging can result in improved **reliability** as messages can be stored until they can be delivered.
- Messaging can be used to **spread load over time**. Messages may heap up in high-demand spikes, but the backlog can be worked off during periods of lower demand.
- Messaging can simplify **processing resources optimization** as allows for optimizing the number of processors for each step in a processing pipeline.
- Messaging can improve **auditability** as one can log the outputs of each processing stage.

### 4.3.2 Why

Benefits of using integration patterns include:

1. They provide simple, proven solution components to re-occurring problems.
2. Often, using one of the integration patterns results in a simplification of the integration solution.
3. Making use of a catalog of integration pattern often speeds up the design process, the process of recovering and understanding aspects of a software architecture, and the process of assessing different integration solutions.
4. Using the standard concepts and terminology simplifies communication between integration experts.
6. Using integration patterns results in more standardized integration approaches which simplifies maintainability.

### 4.3.3 Message channels

Systems or parties integrate via an exchange of artifacts containing information. To this end they require a channel between the parties which can transport the exchanged artifacts.
### 4.3.3.1 Point-to-point channels

In the case of point-to-point channels, each message which is sent along the channel is consumed by only a single consumer. There may still have multiple consumers and the pattern then provides a simple load balancing solution across the different consumers. Note that each message is only consumed once.

![Figure 4.1: The structure of a point-to-point channel.](image)

Figure 4.1 shows the structure of a point-to-point channel.

### 4.3.3.2 Publish-subscribe channels

In the case of publish-subscribe channels, multiple consumers may register to receive messages which are published on the channel. A message is consumed by all consumers which were subscribed to the channel when the message was sent. Figure 4.2 shows the structure of a publish-subscribe channel.

![Figure 4.2: The structure of a publish-subscribe channel.](image)

Some publish-subscribe channels support the concept of *durable subscribers*. A durable subscriber receives all messages published during its subscription period, irrespective of whether they were connected when a particular message was sent or not. The subscription period is thus decoupled from the connection period.

### 4.3.3.3 Datatype channel

The *datatype channel* attempts to address the following problem. When sending different artifact types across a channel, the consumer needs to be able to identify the data or artifact in order to know how to process it. This introduces additional complexities and processing overheads.

The structure of the datatype channel pattern is depicted in figure 4.3. The *Datatype channel* pattern requires different channel for each data/artifact type. This facilitates streamlined processing and enables one to remove any meta-data added to the message to identify the artifact type, as well as the logic required to identify the artifact type.
4.3. INTEGRATION PATTERNS

4.3.3.4 Invalid message channels

The invalid message channel pattern aims to address the following problem. Communicating invalid message back to the message producer might be tricky. Ignoring invalid messages may leave the producer under the false impression that all messages were processed.

In the Invalid message channels pattern (see figure 4.4) the message consumer simply dumps message onto invalid message channel. Other processes may either correct problem and resubmit the message, communicate problem back to message sender, make invalid messages somewhere available to sender for retrieval, or simply log for accountability reasons. In the mean time, the consumer continues processing next message.

4.3.3.5 Dead letter channel

The dead-letter channel pattern addresses a similar problem to the invalid message channel pattern, except that this time the message is valid, but cannot be delivered.
The pattern structure is shown in figure 4.5. The message which could not be delivered is put onto the dead-letter channel where it can be either held for a later attempt for redelivery, logged for auditing purposes or can be made available to the message sender for retrieval.

### 4.3.3.6 Guaranteed delivery channel

The guaranteed delivery channel addresses the following problem. Integrating parties have independent life cycles. A message which is meant for a consumer may not be deliverable when the producer sends it, i.e. the consumer not available or busy.

![Figure 4.6: The structure of the guaranteed delivery channel.](image)

The guaranteed delivery channel (see figure 4.6) maintains the message until either the message is delivered or the consumer acknowledges completion of processing of message. The latter has the advantage that the message will be redelivered if the processing of the message fails.

### 4.3.3.7 Channel adaptors

The problem addressed by the channel adaptor pattern is the following. A provider or consumer may need to communicate with a consumer through some channel. The other party may not be able to use that channel. The producer does not want to support another channel because perhaps wants to stick to some standard, or other parties use that channel and do not want to maintain a separate channel.

![Figure 4.7: The structure of the channel adaptor pattern.](image)

The solution provided by the channel adaptor is depicted in figure 4.7. The adaptor pulls off messages for the subscriber from the channel and feeds these messages to the channel supported by subscriber.

### 4.3.3.8 The message bridge pattern

The problem addressed by the message bridge pattern is the following. Assume multiple messaging systems are used within an organization or system. Different messaging
4.3. INTEGRATION PATTERNS

systems might be based on technology preferences for certain areas. Messages published on one messaging system may have to be available to parties which is connected to another messaging system.

Figure 4.8: The structure of the message bridge pattern.

In the case of the message bridge pattern (see figure 4.8), the bridge is constructed from a range of adapters between the messaging systems. Each adapter pulls messages of one messaging system and pushes them onto another.

4.3.3.9 The message bus pattern

The problem addressed by the message bus pattern is the following. Many systems need to be integrated. Each system needs to know how to integrate with many other systems. This may result in excessive complexity and maintenance costs.

Figure 4.9: The structure of the message bus pattern.

The message bus (see figure 4.9) is a sophisticated message channel which decouples message producers and consumers. It is responsible for:

- providing the connectivity to message producers and message consumers,
- routing messages to appropriate message consumers,
- transforming messages to format as required by consumer, and
- supporting an adapter layer which enables one to plug in an adapter for each system.

Any system only needs to know how to communicate with bus.

4.3.4 Message construction patterns

Message construction patterns represent best practices design components for constructing messages.
4.3.4.1 The command message pattern

The problem addressed by this pattern is the following. A recipient receives a message, but may not know what to do with the message. Also, one may not want to follow a Remote Procedure Call (RPC) approach.

![Command Message Pattern Diagram](image)

Figure 4.10: The structure of the command message pattern.

Figure 4.10 shows the structure of the command message pattern. The command message itself contains not only the information/artifacts which are to be transmitted, but also the request itself, i.e. what to do with message. Command messages are usually applicable to point-to-point channels.

4.3.4.2 The document message pattern

The document message pattern addresses the following problem. Assume you have a message, but you do not know what to do with it – that is the responsibility of the message consumer, i.e. you want to leave the decision of what to do with some information to the message consumer.

![Document Message Pattern Diagram](image)

Figure 4.11: The structure of the document message pattern.

In the case of document messages (see figure 4.11) the message contains only information — not what to do with the information. Document messages are commonly applicable to publish-subscribe channels.

4.3.4.3 The event message pattern

An event is the occurrence of some thing which is in some ways significant. However event sources should not need to know who processes the event and what they are going to do with it.

The event message pattern (see figure 4.12) avoids tight coupling by introducing an event channel. Events are put onto the event channel, but the process which puts
4.3. INTEGRATION PATTERNS

4.3.4.4 The request-reply channel pattern

The problem addressed by the request-reply channel pattern is the following. Often senders require a response to a message. But wants to use a messaging infrastructure and not a synchronous communication channel. This may be because one wants the improved decoupling, the ability for the producer to continue with other work in the mean time or that the consumer does not want to be exposed to the the producer still being available when the response is ready to be dispatched.

In the case of the request-reply channel pattern one constructs Have channel for request messages, and one channel for reply messages.

4.3.4.5 The correlation identifier pattern

The correlation identifier pattern addresses the following problem. Client sends a number of requests related to different processes. When receiving the responses, they need to somehow pair responses up with the original requests.

In the case of the correlation identifier pattern a the producer adds a correlation identifier to request messages which uniquely identifies requests. the consumers use that same correlation identifier on their response messages enabling the original requester (producer) to match responses to requests.
4.3.4.6 The message sequence pattern

The problem addressed by the message sequence pattern is the following. Messages may, at times, be very large. It may not feasible or efficient to send such large message over channel.

The solution provided by the message sequence pattern splits messages into parts. Each message part contains
- a sequence identifier indicating to which sequence the message belongs and
- a sequence number so that the original message can be correctly re-assembled.

This information enables the message recipient to identify the message a particular part belongs to and to re-assemble the message from the parts.

4.3.4.7 The message expiration pattern

The message expiration pattern addresses the following problem. Messages which are not consumed may continue to consume increasing amount of resources (e.g. storage resources). Messages which are not processed within some period might be useless, e.g. storm warning after the storm. The message channel or even recipient may not be in a position to decide whether a message may be expired. Often only the sender knows when a message may be expired.

When using the message expiration pattern, the message producer adds an expiry date/time to the message. The message channel then knows when it may expire/drop the message.

4.3.4.8 The format indicator pattern

The problem addressed by the format indicator pattern is the following. A message channel may be used to send messages in different formats. It might be non-trivial
4.3. INTEGRATION PATTERNS

4.3.5 Routing patterns

*Routing patterns* provide reusable design solutions which

- route message to appropriate/suitable consumer,
- decouple message producers from message producers,
- distribute messages across consumers,
- filter messages,
- aggregate or split messages and
- perform load balancing.

### 4.3.5.1 Content based routing

The problem addressed by the *content-based routing* pattern is the following. A message sender should often not know who should process a message. Consider the example, of a flight booking and meal request submitted to airline. The flight booking would have to be processed by reservations whilst the meal request is processed by catering. The client should not have to know where to send the requests to, but simply submits all requests made to the airline to a single access channel provided by the airline.
A content-based router investigates content of message and routes the message to the appropriate message processor. The pattern fully decouples message senders from actual message processors.\footnote{In pipes & filters / SOA architectures one service has no knowledge of which is next processing step. Requires content based routing.}

### 4.3.5.2 The message filter pattern

The message filter pattern aims to address the following problem. A large number of messages may be published on a particular channel. A particular consumer might, however, not be interested in all the messages. Consider, for example, a share price channel which publishes all share price changes. A particular consumer might be interested in only share price changes for a particular entity or sector.

![Figure 4.18: The structure of the message filter pattern.](image)

The message filter performs one or more checks on the message content and/or the headers. It passes only messages through which satisfy condition. Filters can be logically combined using the logical AND, OR, XOR and NOT operators.

### 4.3.5.3 The dynamic router pattern

The dynamic router pattern addresses the following problem. At times the message recipients need to be selected dynamically, i.e. in real time. This could involve selecting current best service provider on, for example, the basis of cost, performance, or reliability. Alternatively message recipients could be dynamically selected for the purpose of load balancing.

![Figure 4.19: The structure of the dynamic router pattern.](image)

A dynamic router stores selection rules in rule base. The rules are evaluated to determine whether a message is passed on or not. The dynamic router routes the message to the first recipient for which the rules are fulfilled.
4.3. INTEGRATION PATTERNS

4.3.5.4 The recipient list pattern

The problem addressed by the recipient list pattern is as follows. A client may want to send a message to multiple recipients. However, the client does not want to manage the process and occur overheads of sending individual messages to each recipient.

![Recipient List Pattern](image)

Figure 4.20: The structure of the recipient list pattern.

In the case of the recipient list pattern one defines a channel for each recipient. It then uses a recipient list router which inspects the incoming message, determines the list of desired recipients, and forwards the message to all channels associated with the recipients in the list.

4.3.5.5 The splitter pattern

The problem addressed by the splitter pattern is the following. A message may contain a range of different types of information. Different parts of the message to be processed by different processors.

![Splitter Pattern](image)

Figure 4.21: The structure of the splitter pattern.

Splitters split messages into individual components and place those components onto different channels for different processors.

A special case is a batch splitter which splits a batch into individual requests or messages.

4.3.5.6 The aggregator pattern

The problem addressed by the aggregator pattern is the following. Information received from multiple parties needs to be combined into a single message to be forwarded to some message consumer.

The aggregator collects and stores the individual messages until a complete set of related messages has been received. It then constructs a composite message from these individual messages and publishes it on appropriate channel.

A batch aggregator is a special case of this pattern. It assembles responses to set of requests into a batch of responses.
4.3.5.7 Resequencer

The problem addressed by the resequencer pattern is the following. A message router may route sequence of messages along different routes. The messages received through different routes may no longer be in sequence.

A resequencer collects and re-orders messages. The messages are then published in correct order.

**Note:** A resequencer does not have to wait for the full sequence to be received before forwarding sequence components.

4.3.5.8 The scatter-gather pattern

The problem addressed by the scatter-gather pattern is the following. Assume you need to source multiple quotes for each order item. Once you have them you need to make a decision on which quotes are best and assemble actual order.

When using the scatter-gather, one broadcasts a message to multiple recipients. One then uses an aggregator to aggregate the responses back into a single message.
4.3.9 The composed message processor pattern

The composed message processor pattern aims to address the following problem. Assume you need to perform a range of processing steps on message components. Once done, you need to forward full message.

For example, assume you need to check availability of each order item with stock management. Once availability of all order items established, need to continue processing order.

Figure 4.25: The structure of the composed message processor pattern.

The composed Message Processor splits the message up, and routes the sub-messages to the appropriate processors. It then re-aggregates the responses back into a single message.

4.3.10 the routing slip pattern

The routing slip pattern addresses the following problem. Assume a message needs to go through a sequence of processing steps. Individual processors should not have to know what the next processing step is.

When using the routing slip pattern one attaches a routing slip to each message, specifying the sequence of processing steps. Each component is then wrapped with a special message router that reads the routing slip and routes the message to the next component in the list.

Figure 4.26: The structure of the routing slip pattern.

4.3.11 The process manager (controller) pattern

The routing slip pattern assumes that the sequence of processing steps can be determined up-front and that the processing is to be done through sequence (linear) of processing steps.

The process manager pattern or controller pattern addresses the following problem. What should one do when the processing sequence depends on outcomes of other
processing steps, and if certain processing steps can be executed concurrently. One still wants to ensure that processors do not need to know where to route a message next.

![Diagram of process manager (controller) pattern](image)

**Figure 4.27: The structure of the process manager (controller) pattern.**

When using the *process manager* or *controller* pattern one localizes the work-flow logic within a process manager or controller. The process manager maintains process state and determines the next processing step based on intermediate results.

### 4.3.5.12 The message broker pattern

The *message broker pattern* addresses the following problem. Assume you want to decouple the destination of a message from the sender, yet you want to maintain central control over the flow of messages.

![Diagram of message broker pattern](image)

**Figure 4.28: The structure of the message broker pattern.**

The *message broker pattern* is based on the hub-and-spoke architectural style. Messages are received from multiple destinations. The broker determines the message destination and routes the message to the correct channel.

### 4.3.6 Transformation patterns

Transformation patterns transform the content and/or meta-data of a message en-route.
4.3. INTEGRATION PATTERNS

4.3.6.1 The envelope wrapper pattern

The envelope wrapper pattern addresses the following problem. Assume you need to communicate extra information required by infrastructure and certain message processors. You would not like to add this information to the message since you do not want to pollute the message and since it might pose problems for message processors.

![Figure 4.29: The structure of the envelope wrapper pattern.](image)

When using the envelope wrapper pattern, one wraps the application data inside an envelope that is compliant with messaging infrastructure. The message is unwrapped when it arrives at the destination. Message envelopes are commonly used to communicate meta-data around messages including correlation ids, authentication credentials or user roles, transactional contexts, information about response channels or formats, and so on.

4.3.6.2 The content enricher pattern

The problem addressed by the content enricher pattern is the following. A service provider may require more business information than client can provide. Neither source system nor target system has access to the additional required information.

![Figure 4.30: The structure of the content enricher pattern.](image)

A content enricher obtains the additional information from external data source and adds the additional business information to core message body.

4.3.6.3 The message normalizer pattern

The message normalizer pattern addresses the following problem. Assume different systems use different protocols. You may need to convert between all these protocols. The combinatorics results in high complexity. Also, core components may have to operate on messages in different protocols. Different messages may be semantically equivalent, but may arrive in different formats.
CHAPTER 4. INTEGRATION ARCHITECTURE

4.3.6.4 The content filter pattern

The content filter pattern addresses the following problem. A message may have a lot of information. Not all information required for down-stream processing.

A content filter removes any unnecessary information from the content. This reduces communication overheads as well as down-stream de-marshaling and processing complexity.

4.3.6.5 The claim check pattern

The claim-check pattern aims to address the following problem. A message has content not required by some intermediate processing steps or that the intermediate processing units should not see some of the content of a message which is to be processed.

When using the claim check pattern one receives a message, extracts message elements which either are not required for some intermediate processing steps, or should not be visible for part of the processing pipeline, generates a unique key for content which is to be temporarily removed, persists content to be removed with key, and replaces
that content in message with unique key. At a later stage the key is used to retrieve the persisted content and inserts it back into message.

4.3.7 End-point patterns

Message endpoint patterns are patterns for message senders and receivers. They represent best practices for the structure and behaviour of message end points and cover synchronous and asynchronous communication, pull and push paradigms and indempotency.

4.3.7.1 The message mapper pattern

The problem addressed by the message mapper pattern is the following. Assume you need to convert domain objects to messages as required by message channel. However, domain objects and messaging channel should not know of each other.

![Figure 4.34: The structure of the message mapper pattern.](image)

The pattern suggests to create separate message mapper which contains the mapping logic between the messaging infrastructure and the domain objects. Note that neither objects nor infrastructure have knowledge of message mapper’s existence.

4.3.7.2 The message gateway pattern

The problem addressed by the message gateway pattern is the following. Assume you want to have systems integrate via messaging in a type-safe way, without the system knowing they are using message based integration. Systems should request normal business services.

![Figure 4.35: The structure of the message gateway pattern.](image)

A messaging gateway wraps messaging-specific method calls and exposes domain-specific methods to the application. The pattern keeps messaging logic separate from application logic.
4.3.7.3 The polling consumer pattern

The problem addressed by the polling consumer pattern is the following. Assume an application wants to consume messages, but wants to control when it does so. The application needs to know when message is available, but messaging infrastructure does not support events.

Figure 4.36: The structure of the polling consumer pattern.

A polling consumer polls the messaging infrastructure for available message. The application then requests message from polling consumer when ready to process message. The polling consumer may generate events notifying the application that messages are available.

4.3.7.4 The event-driven consumer pattern

The event-driven consumer pattern addresses the following problem. Assume a message processor needs to consume a message as soon as it is delivered.

Figure 4.37: The structure of the event-driven consumer pattern.

Event-driven consumers have a messaging infrastructure which generate events on message receipt and the message consumer has an event handler which can directly process event.

Note that events may contain message data. The pattern reduces the communication overheads by removing polling requirement at the cost of having to support a more sophisticated messaging environment.

4.3.7.5 The competing consumers pattern

The problem addressed by the competing consumers pattern is the following. Assume a single consumer cannot process messages fast enough. We require concurrent message processing but have no load balancing infrastructure.

When using the competing consumers pattern, one inserts messages into a point-to-point channel and registers multiple consumers with that channel. Each consumer processes a next message as soon as soon as they have capacity.
4.3. INTEGRATION PATTERNS

The pattern provides an concrete form of processor (service-provider) driven load-balancing. It also improves reliability by providing fail-over safety through the multiple consumers. The pattern provides an efficient mechanism for load balancing without requiring a load balancer.

### 4.3.7.6 The selective consumer pattern

The problem addressed by the *selective consumer pattern* is the following. Assume a message processor is assigned to consume messages off a channel and that the consumer should not process all messages, just some of them. How can a message consumer select which messages it wishes to receive?

The *selective consumer pattern* requires that the consumer filters messages delivered by its channel, extracting only messages that fit some criteria from the channel. The pattern is commonly used for prioritized processing with a number of processors assigned to process high priority messages only.

### 4.3.7.7 The message dispatcher pattern

The problem addressed by the *message dispatcher pattern* is the following. Assume you need to have multiple message consumers coordinate their message processing. Different types of messages should be consumed by different consumers. For example, all messages for a client should be dispatched to the appropriate client.

A *message dispatcher* consumes messages from a channel and distributes them across processors.
4.3.7.8 The indempotent receiver pattern

The indempotent receiver pattern addresses the following problem. Assume that due to quality of service issues on the delivery channel or any other reasons a particular message may be received multiple times by a message processor. However, messages should be processed only once.

In the case of the indempotent receiver pattern, one has a unique identifier assigned to each message. Indempotent receivers keep track of the message identifiers of the messages they have thus far received. Any duplicate messages are dropped.

4.3.7.9 The service activator pattern

The problem addressed by the service activator pattern is the following. Assume you want to have service which can be triggered via different messaging technologies, and via direct synchronous requests.

A service activator extracts messages from channel and makes an appropriate method call or service request for that message. The pattern can be used for request-reply or one-way scenarios.

4.3.7.10 The durable-subscriber pattern

The problem addressed by the durable subscriber pattern is the following. Assume a message receiver may go off-line sometimes. The receiver may not want to not miss
any messages which were published whilst not connected.

Figure 4.42: The structure of the durable subscriber pattern.

In the case of the **durable subscriber pattern**, the channel keeps track of the subscription periods of the different subscriber, and maintains separate queue of message references for each subscriber. Messages are only removed from the channel if they are no longer referred to by any subscriber queue. The message is usually combined with the **publisher-subscriber** pattern.

### 4.3.8 System management patterns

System management patterns concern themselves with how to perform technical tasks with minimal impact on business processes, and on the required infrastructure.

#### 4.3.8.1 The control bus pattern

The problem addressed by the **control bus pattern** is the following. How can we effectively administer a messaging system that is distributed across multiple platforms and across a wide geographic area?

Figure 4.43: The structure of the control bus pattern.

In the **control bus pattern** one uses a Control Bus to manage an enterprise integration system. The control bus uses the same messaging mechanism used by the application data, but separate channels to transmit data that is relevant to the management of components.

#### 4.3.8.2 The wire-tap pattern

The **wire-tap pattern** addresses the following problem. Assume you want to use a point-to-point channel because each message should be consumed by single consumer.
However, for testing, monitoring, debugging, want to inspect all messages sent across a channel.

![Figure 4.44: The structure of the wire-tap pattern.](image1)

The **wire-tap pattern** inserts a simple recipient list into the channel that publishes each incoming message to both, the main (business) channel and a secondary (technical) channel. The process flow is not altered.

### 4.3.8.3 The detour pattern

The problem addressed by the **detour pattern** is the following. Assume you need to not only inspect messages, but also to modify or reroute them, i.e. you want to route a message through intermediate steps to perform tasks like validation, testing or debugging.

![Figure 4.45: The structure of the detour pattern.](image2)

The **detour pattern** constructs a detour with a context-based router controlled via the control bus. Depending on the router state, the router routes messages either through additional steps, or directly to destination channel.

### 4.3.8.4 The message store pattern

The **message store** pattern addresses the following problem. Assume you need to provide reporting or auditing against message flows. You do not want to impact on the loosely coupled, transient nature of messaging system.

![Figure 4.46: The structure of the message store pattern.](image3)

In the case of the **message store pattern** one uses a message store to capture information about each message in a central location. One can have separate channels feed into the same message store.
4.3. INTEGRATION PATTERNS

4.3.8.5 The smart proxy pattern

The *smart-proxy pattern* addresses the following problem. Assume you want to track messages, but some services publish reply messages to return address specified by client.

![Figure 4.47: The structure of the pattern.](image)

The *smart proxy* stores the return address supplied by client, and replace return address with address of the Smart Proxy. The return received by smart proxy is forwarded to requested return address.

4.3.8.6 The test-message pattern

The *test-message pattern* addresses the following problem. Assume you are using heart beat messages issued by system components. The messages may contain health, load and other information. However, the heartbeat messages with vital info may be corrupted due to internal fault. How do I know heart beat messages represent actual state of component?

![Figure 4.48: The structure of the test message pattern.](image)

The *test message pattern* generate test data for the component and uses a test separator to route business messages on to business processing units, and and test messages to test validation.

4.3.8.7 The channel purger pattern

The *channel purger pattern* addresses the following problem. At times there are left-over messages in channels. This may effect tests or the running system.

A *channel purger* is used to removes unwanted messaged from channels.
4.4 Best practices

In this section I discuss some general guidelines as well as specific guidelines for request and document based integration and for integration infrastructures.

4.4.1 Some general best practices around integration

Some general best practices you might want to keep in mind include:

1. Stick to standards.
2. Make quality attribute trade-offs explicit
   - e.g. restful web services with JSON ⇔ SOAP-based web-services with XML schema
   - usability & simplicity versus quality control & pluggability
3. Prefer document-based over request-based integration
   - weaker coupling.
   - localizes responsibilities where they should be.
4. At the low-level, use a components-based approach.
5. At the higher-level, use either a services-oriented, messaging or observer

4.4.2 Best practices for request-based integration

Some best practices around request-based integration include

1. Make service providers/components discoverable through registering them with naming and trader services.
2. Use dynamic service discovery
3. Use dependency injection for improved flexibility, testability and to ensure good responsibility localization.
4. Decouple via services/component contracts whose specification include the specification of pre- and post-conditions, quality requirements as well as the data structure for exchanged objects. Preferably use versioning with your contracts.

4.4.3 Best practices for document-based integration

You might want to consider the following best practices around document-based integration:
4.5. EXERCISES

1. Decouple via data-structure specifications
   - Use standards like XML schemas (closed world) and RDF/OWL schemas (open world) for the data structure specification.
2. For flexible, we may move toward self-describing with semantics (e.g. OWL schema).

4.4.4 Best practices for integration infrastructure

When designing an integration infrastructure, you may want to consider including some of the following:

1. externalizing integration aspects (remove from systems you are integrating) using a services bus, a messaging bus or an observable space,
2. Service/component registries with appropriate lookup services,
3. Including a dependency injection framework,
4. Providing contract repositories for publishing and sourcing service or component contracts.
5. Maintain integration policies including the policies around accredited and/or preferred service providers.

4.5 Exercises

1. Considering a system you have been working on which required extensive integration
   a) Identify the integration approaches and mechanisms used, how they were concretely implemented and analyze the rationale for using them.
   b) Identify any channel patterns used within the system and discuss the rationale for using those patterns.
   c) Identify any message construction patterns used within the system and discuss the rationale for using those patterns.
   d) Identify any routing patterns used within the system and discuss the rationale for using those patterns.
   e) Identify any transformation patterns used within the system and discuss the rationale for using those patterns.
   f) Identify any end-point patterns used within the system and discuss the rationale for using those patterns.
   g) Identify any system management patterns used within the system and discuss the rationale for using those patterns.
Chapter 5

Reference architectures

5.1 What is a reference architecture and a framework

A reference architecture is defined as a best-practices based template architecture which has been proves to address the typical challenges for a particular domain. It provides a specific combination of architectural patterns, and supports a range of architectural strategies in order to concretely realize quality requirements. In addition a reference architecture commonly either relates to or specifies a set of standards.

A framework, on the other hand, is defined as a partial or complete implementation of a reference architecture.

A framework thus provides an implementation of the infrastructure of the reference architecture based on the architectural patterns used by the reference architecture, and each of the architectural strategies specified in the reference architecture.

5.2 Examples

A number of reference architectures are widely used. Some are reference architectures for enterprise systems, but there are reference architectures for fields as disparate as vehicle control systems and gaming. Some of the more widely used reference architectures include

- Layered reference architectures for enterprise systems like Java-EE and Python Django. These reference architectures typically emphasise scalability, security, reliability, and, to a lesser extend, integrability. They introduce concepts for presentation, services and persistence layer business components. Java-EE has many implementing frameworks like JBoss, Apache Geronimo, and Glassfish as well as a range of commercial Java-EE implementations.

- The Services-Oriented (reference) Architecture which places the core focus on integrability, felxibility (time to market) and re-use. It is based on the microkernel pattern and introduces the concept of a stateless, discoverable and self-healing
service as core concept for specifying application logic. In addition it provides an infrastructure for “service orchestration”. Once again, there are many implementing frameworks like Apache Axis, OpenESB, Mule and so on.

• *Space-Based Architectures* which focus on extreme reliability and scalability, integrability and the ability to have workers auto-orchestrate a process amongst themselves. This reference architecture is based on the blackboard architectural pattern.

• *AUTOSAR* is a widely used reference architecture for automotive systems. It is based on the layered and microkernel architectural pattern and focuses on addressing reliability, performance, integrability and monitorability. Once again, there are many implementing frameworks, most of which are proprietary implementations of this open reference architecture.

### 5.3 Benefits and risks of using reference architectures

Benefits of using a reference architecture include

1. *Reduced risk*
   - Community contributes to solution
   - Typical architectural challenges for domain addressed
   - Well understood by architects and developers
   - Often enforces good standards compliance.

2. *Lower cost*
   - Lower analysis and research costs.
   - Implementing frameworks commonly exist.
   - Lower maintenance costs because frameworks often maintained by community.
   - Lower training costs for staff and shorter time to productivity.

However, having a software architecture on a reference architecture also introduces some risks. In particular that of jumping to reference architecture prior to solid architecture analysis, and that it may curb innovation/competitive advantage.

### 5.4 CORBA

#### 5.4.1 What is CORBA?

CORBA, the Common Object Request Broker Architecture is a reference architecture for object-oriented integration independent of implementation technologies, location, network technologies, and protocols. It is an object-oriented middleware with well over 200 implementations. Common uses of CORBA include that of providing a base infrastructure for enterprise systems (e.g. a base technology for Java-EE), and providing a high-performance, reliable integration infrastructure for telecommunication, defense, vehicle control and many other systems.
5.4.2 Overview of CORBA reference architecture

Figure 5.1: Overview of the CORBA reference architecture.

5.4.3 The CORBA object model

A CORBA object is a user object which may or may not ever be represented by a physical object. For example, the physical system might be a Cobol system which does not have any objects and the CORBA framework does the object-procedural mapping. But even in technologies which support objects, the user object may be represented by different physical objects over time. This is so since CORBA object model implements the flyweight pattern with objects being passivated to some temporary or permanent persistent storage and activated again at some later stage. The client code thus has a reference or handle to a to virtual user object and not to a physical object.

In CORBA a component contract is represented by a technology neutral interface specified using CORBA’s Interface Definition Language (IDL). This IDL can be mapped onto different technologies (e.g. a Java interface).

5.5 Java-EE Architecture

5.5.1 Java-EE: first level of granularity

The application server is the architectural component at the first level of granularity. It is based on the layered architectural pattern with the following layers
1. Access layer → web container, message queues, …
2. Business processes layer → EJB container
3. Persistence layer → Persistence context

A number of tactics or strategies are used to concretely address the quality requirements of a typical enterprise system. For example, clustering of all architectural components including all layers, JNDI repositories, load balancers, firewalls and databases is used to address scalability, availability and reliability.

At the first level of granularity no application components are hosted and hence the architecture thus does not introduce and concepts or constraints for application components at this level of granularity.

The EJB container is the business logic container.

At this lower level of granularity a further range of architectural tactics is applied to address EJB container quality requirements. For example, the flyweight pattern and resource pooling (including object, thread and connection pooling) are used to address scalability and performance. Similarly interception is used to address for auditability...
and security, and queueing to address scalability and reliability. In addition, integrability is addressed through seamless integration through RMI, SOAP-based and REST-ful web services, and messaging.

The EJB container does host application components. It uses the controller pattern for application components and introduces stateless and stateful session beans and entities as the concepts within which applications are developed. Message driven beans should really be used for messaging adapters and are thus conceptually part of the access layer.

In addition EJB introduces a number of constraints for application components (session beans) including

- may not provide no direct access to service components
- may not directly access resources (threads, DB connections, . . .)
- may not directly access native libraries

Finally EJB introduces a range of strategies to improve flexibility, reliability and extensibility including

- decoupling via contracts
- dependency injection & service provider lookup
- declarative authorization
- declarative transaction management
- interception for extensibility and customizability

**Scalability**
- Typically good through resource (thread, connection, object, . . .) reuse, load-balancing, caching and clustering.

**Reliability**
- Typically good through transaction support, clustering, session replication and support for messaging.

**Flexibility**
- Average as processes not explicit.
- Improved through interceptors, removal of infrastructure/plumbing logic, hosting process logic only in stateless session beans and annotations.

**Performance**
- Average because of layers, and communication overheads.
- RMI/IIOP is reasonably efficient protocol.
- Improved through JPA-based object cache.

**Auditability**
- Not directly supported. Needs to be implemented via interceptors.

**Security**
- Good support for authentication, authorization and confidentiality.

**Integrability**
- Quite good with support for CORBA, SOAP-based & Restful web services, DB integration.
- Integration to systems using proprietary protocol via JCA.
5.5.2 What does the EJB container provider?

5.5.2.1 Introduction

As the EJB container is responsible for hosting shared business objects within the application server, it is responsible for transparently applying a number of enterprise and middleware services to beans. Had the developer needed these services in a traditional CORBA/DCOM or Java RMI deployment, several explicit API calls would have to be made from within the business objects. In other words, the business logic would be polluted with a number of complex technicalities.

5.5.2.2 Concurrency support

The application server automatically supports concurrent service requests via multiple bean instantiation, i.e. concurrent service requests are processed by different bean instances. Bean developers need thus not worry about writing multi-threaded servers – in fact they are forbidden to do so because that would interfere with the containers concurrency support.

Furthermore, in the case where the different client bean instances share common data resources the container takes over the responsibility of synchronizing the different data views.

5.5.2.3 Component pooling

Component pooling is possible because, as we shall see, clients do not obtain a direct reference to the enterprise bean instance.

As one deploys enterprise beans within a container, the container typically creates a pool of instances of the bean. The algorithm used is specific to the container – this is one of the many areas where application server vendors compete.

Pooling is particularly simple for stateless session beans which provide a set of client services but which do not maintain information across service requests. In this case the session bean is returned to the pool upon completion of the service and if the same or another client requests a service from that same enterprise bean one of the bean instances in the pool is allocated to the client.

Containers also provide component pooling and life-cycle management for other enterprise beans. For message-driven beans it is basically as simple as for stateless session beans. For stateful session beans and entity beans the state has to be persisted before the same bean instance can be used for another client.

If, say during peak hours, the demand for a particular enterprise bean increases, the container can dynamically increase the pool size and at a later stage, when the load decreases, the pool size can be reduced again.

A relatively small number of beans can thus serve a large number of clients.

5.5.2.4 Network enabling

Bean developers need not make the beans network-enabled. The container automatically supports distributed architectures by wrapping the bean instances by bean ob-
jects which are network enabled. These bean objects intercept bean service requests, in order to provide the other services like transaction, security and persistence. Every EJB is published as both an RMI and a CORBA object, and can easily (through the use of Java annotations) be published as a Web Service as well.

5.5.2.5 Persistence

The application server automatically loads bean information from persistent storage upon bean activation and saves the bean state automatically onto persistent storage upon deactivation.

Furthermore, the bean developer need not know the details of the structure of the persistent storage (e.g. which object fields are stored in which columns of which tables or whether the persistent storage is a relational or object database). The mapping to persistent storage can be done declaratively by a database administrator who need not be a Java developer. It can be even left to the EJB container to create the required database tables and to define the mapping implicitly. **Note:** If one uses JPA based persistence, then one is locked into object-relational mapping and hence to relational databases.

EJB also maps specialization relationships onto relational databases and allows you to choose from a set of standard mappings for specialization.

5.5.2.6 Component location transparency

EJB Application Servers must provide a naming and directory service which implements the Java Naming and Directory Interface (JNDI). The JNDI is a generic API for interfacing with general naming and directory services (such as, for example, LDAP).

As is shown can be seen in 5.3, JNDI wraps concrete naming and directory services. Some of the naming and directory services which can currently be accessed through JNDI are

- **Local File Systems:**
  Files and directories

- **COSNaming:**
  CORBA’s standard naming service is meant to enable CORBA clients to look up a reference to a CORBA object from a name.

- **RMI registry:**
  Java’s RMI naming service which fulfills the same purpose as the CORBA naming service for Java RMI objects.

- **LDAP:**
  The *Lightweight Directory Access Protocol* was developed in the early 1990’s as a standard directory protocol which would be used by a wide range of applications. It facilitated, for example, that the particulars (e.g. personal details, phone numbers, e-mail addresses, network and device access particulars, ...) of a new employee could be entered into one central location. Applications like phone and e-mail number search applications, answering machine services, network
administration applications etc. would all obtain the required information from one central LDAP. LDAP data is structured as a hierarchical database which allows multiple entries for a specific item. Sun’s iPlanet directory server and OpenLDAP are the most well known LDAP servers but most other directory services provide an LDAP interface. LDAP version has support for referrals – i.e. it makes it possible that LDAP service requests are referred on to other service providers. This enables large-scale clustering and distribution.

- **DNS:**
  Internet domain name servers which map a domain name (the host name) onto a IP address (the message path) can also be accessed through JNDI.

- **NIS:**
  NIS, Sun’s Network Information Service, which acts as a yellow pages service for network resources.

- **NDS:**
  The Novell Directory service.

JNDI thus provides a standard interface which decouples the application from the physical naming and directory service provided by the environment. Naming services are typically used to obtain a reference to an object in a distributed environment. Directory services are really sophisticated naming services which include metadata describing the objects they reference. This enables clients to make more sophisticated searches for objects – i.e. for example to query all printers in a particular building which can print color onto A3 sized paper.
5.5.2.7 Transaction support

A transaction is a set of operations that must be processed as a single unit and if that unit was not successful in its entirety the entire transaction must be rolled back. Transaction boundaries are used as instants where object states are synchronized with the database.

The EJB application server/container provides implicit support for distributed transaction management freeing developers of the burden of either including API calls to managing transactions themselves.

5.5.2.8 Security support

When you open your systems across an intranet or even internet, security becomes of vital importance. The EJB application server facilitates a declarative support for authentication and authorization which removes the burden of making calls to a security API from the bean developer.

The authentication is usually done at the persistence layer using normal JAAS (Java Authentication and Authorization Service) based security.

Confidentiality is usually transparently configured within the architecture resulting in secure communication (via, for example, SSL).

At the services layer, the main focus is on authorization. The bean deployer defines security roles for entire enterprise beans or for individual services supplied by enterprise beans. A security administrator maps users and user groups onto security roles.

5.5.2.9 Session management

The client session is solidly managed across presentation and services layers (the session beans). The session context and state is propagated across session beans.

5.5.2.10 Interception

Enables you to intercept both, business logic and bean management services in order to add further responsibilities around a set of base responsibilities addressed by the bean logic itself and by the application server. This makes enterprise beans externally extensible.

5.5.2.11 Resource connection pooling

In a similar way, the container is typically responsible for resource connection pooling like, for example, database connection pooling. Establishing these connections is typically expensive. Enterprise beans should typically not establish connections themselves to resources. Instead they obtain a connection from the container who maintains a connection pool.
5.5.2.12 Implicit monitoring

EJB containers typically monitor the usage of beans in order to

- Optimize bean and thread pools.
- Perform load balancing across multiple machines
- Support reporting for administration and maintenance purposes.

5.5.3 Enterprise beans

Enterprise beans are meant to be pure server side business logic components which can be deployed on application servers within different business processes (work flows) requiring different security and transactional support, as well as different persistence mappings. To achieve this a bean implementation should not contain any deployment information and should focus purely on business logic.

5.5.3.1 Bean species

5.5.3.1.1 Session beans

5.5.3.2 Elements of enterprise beans

5.5.3.2.1 EJBObject  If incoming service requests were dispatched directly to a bean, all of the enterprise services provided by the EJB container would be by-passed. To this end, beans which are deployed in a EJB container are never directly accessible: The EJB container generates a class (which implements both the remote interface of the bean, as well as the javax.ejb.EJBObject interface). The EJB container uses the information specified either as Java annotations, or in XML deployment descriptor, to generate an EJBObject which enforced the qualities desired by the developer (security, transactional behaviour, web services publication, etc).

The EJB object thus intercepts the call to the enterprise bean, enabling the container to provide standard support to your beans, such as

- concurrency,
- transactions,
- security, and
- persistence.

The way in which a service request is intercepted and ultimately processed is shown in the following figure. Note: Since EJB uses RMI/IIOP/TCP/IP as protocol, enterprise beans can be accessed from CORBA through the standard CORBA protocol, IIOP. This means they are directly accessible to, for example, CORBA or C++ clients.

5.5.3.2.2 Local and remote interfaces
5.5.3 Bean restrictions

The container takes over a lot of responsibilities which would otherwise have resided with the bean developer. Consequently the bean itself is prevented from doing certain things which interfere with the container operations.

5.5.3.1 Beans can’t give clients direct access to the bean instance The client should never interface with the bean instance directly, but should instead interface only with the EJB object which is generated by the container and acts as a portal to the bean. This architecture enables the container to intercept bean service requests and take over the responsibility of container managed services like transactions, concurrency, security and persistence.

Though you can’t pass a handle of the bean instance to the client, you can, of course pass the handle to bean helper classes (which are, from the client’s point of view part of the bean implementation).

5.5.3.2 Enterprise beans may not accept network server connections Beans should not act as servers themselves. Once again, this is the responsibility of the EJB object. The bean can, however, open client connections to other network servers (e.g. CORBA servers or other enterprise beans).

5.5.3.3 Enterprise beans should be single-threaded In order to be able to effectively handle concurrent service requests from a large number clients the application server uses thread pooling. For this reason the application classes should not create their own threads. If they do want to have a piece of work done in a separate thread, then they should submit the piece of work to the application server which will assign a thread from a thread pool for processing that piece of work in a managed way.

The bean should also not use synchronization itself. This too will be handled by the application server within the transactions framework.
5.5.3.3.4 **Enterprise beans should not create a user interface** The whole idea behind Enterprise Java Beans is to separate the presentation layer from the business logic layer. No direct interaction via a GUI (AWT or Swing) or keyboard input is allowed for enterprise beans. Of course, the latter follows from the fact that enterprise beans may not use the `java.io` package. Bean developers may also not assume that the bean host has any form of GUI support.

5.5.3.3.5 **Stateful session beans can’t have persistent class fields** The idea of maintaining persistent class information (static fields) for an EJB goes largely against OO concepts and warrants a redesign with perhaps introducing further entity beans.

There are several problems with maintaining class state for an EJB. Firstly, the container will not manage concurrent access to such fields and you as bean developer may not. Furthermore, the EJB service requests may be distributed by the container across Java Virtual Machines (JVMs) and the class state would not be available across these JVMs.

Thus, you should only use constant (`final`) class attributes for EJBs.

5.5.3.3.6 **Enterprise beans may not use any native libraries** The reasons for this are firstly security and secondly portability. If you really need to obtain access to native libraries you should wrap them as a CORBA component and use them via standard CORBA service requests.

5.5.4 **Entity objects**

The Java EE specification supports the concepts of entity objects, which are managed by an entity manager. Entity objects exist from when they are created until they are deleted. The entity manager manages the persistence to durable storage (some form of database).

Entity objects have persistent object identity and can be looked up on that object identity. Hence, these objects exist, from the user’s side, across client sessions and server restarts. The entity manager will typically use JPA (the Java Persistence API) to interface with an object-relational mapper (such as Hibernate or Eclipselink) in order to persist the state of objects to a relational database.

**Note:** Although possible (via JDBC), Java EE proposes that developers no longer work with relational databases using relational elements such as SQL and result sets. Instead, it is proposed to stay within a fully object-oriented realm, and let the infrastructure manage the mapping between the two worlds.

5.6 **Services-Oriented Architectures (SOA)**

Services-Oriented Architectures are widely used within organizations to provide core integration architecture between various systems and to facilitate the orchestration of processes across the various systems used within the organization as well as the service provider and partner systems whose services are integrated into the business processes.
5.6.1 What is a services oriented architecture?

A Services-Oriented Architecture is generally defined as follows:

**Definition 5.6.1.** A Services Oriented Architecture is a software infrastructure within which stateless services can be published, discovered, routed to currently selected service provider, integrated, monitored and managed, and higher-level services can be orchestrated from lower level services.

There exist a wide variety of SOA frameworks, both open-source as well as proprietary. For Java there is a refined specification for SOA which is the *Java Business Integration* (JBI) specification which introduces a normalized message bus, pluggable service engines and pluggable binding components for a wide variety of integration channels including SOAP, REST, RMI, CORBA, SMTP and so on.

5.6.2 Aims of Services-Oriented Architectures

The aims of a Services-Oriented Architecture include

- providing integration infrastructure, decoupling systems and technologies,
- reducing the time-to-market and increase flexibility,
- to expose business processes to be observable and modifiable by business,
- to provide an infrastructure for service reuse across technologies and business units, and to
- provide an infrastructure for service provider and service provision management and governance.

5.6.3 Overview of the SOA reference architecture

![Figure 5.5: Overview of the SOA reference architecture.](image-url)
5.6.4

The SOA infrastructure contains a number of core components addressing architectural responsibilities. The infrastructure between these components is constrained by architectural patterns and architectural tactics are used to address quality requirements.

5.6.4.1 Service registry

A service registry acts as a naming and directory service for web services, i.e. can look up a service by name or by some service classification. It also acts as a contract repository, hosting the WSDL contract specifications. The service registry provides governance tools to tag services as accredited or preferred service providers.

Service registries are commonly packaged together with a service repository, though the latter can also be stand-alone. A service repository hosts the higher level service orchestrations as specified in some orchestration language like BPEL, WS-BPEL or BPMN. It also works in conjunction with the service registry to provide services to assess whether an orchestrated service’s dependencies are met.

5.6.4.2 Service Management Infrastructure

Services-Oriented Architectures (SOAs) specify an infrastructure for service management and service governance. This includes infrastructure for

- service life cycle management,
- static and dynamic service provider selection,
- service access control (authorization),
- service provision oversight (contract oversight), and
- being able to verify whether the dependencies of a service are met.

5.6.4.3 Service containers

Service containers host leaf or atomic services, i.e. services which are not orchestrated from lower level published services. Examples of services containers Spring or EJB containers, XSLT engines, web service or CORBA wrapped systems, Microsoft.Net, ...

5.6.4.4 Process engines

Process engines can interpret and execute process execution engines. Can interpret process execution languages like BPEL, WS-BPEL, and BPMN. Examples of process execution engines are JBoss jBPM, Petals BPEL Engine, Activiti, OW2 ORchestra, as well as a range or proprietary engines.
5.6.1 SERVICES-ORIENTED ARCHITECTURES (SOA)

5.6.4.5 SOA infrastructure patterns

The first level of granularity pattern is SOA is the microkernel with the microkernel itself being the services bus. The lower level components in a SOA architecture are based on the adaptor, broker and hierarchical patterns (the adaptors which in turn use the broker patterns and the services registry which is based on the hierarchical pattern). At lower levels of granularity, SOA uses, for example, the adaptor, broker. SOA also uses a range of integration channels including point-to-point channels, content based routers, message bus and so on.

5.6.4.6 Architectural tactics used in SOA

- Messaging
  - for reliability & decoupling
- Dynamic service provider lookup
  - for flexibility and decoupling
- Interception
  - for auditability (logging), security, . . .
- Clustering of all architectural elements
  - for scalability and reliability
- Compensating work flows to undo on failure.
  - for reliability i.e. leave in consistent state (ACID criteria)
- Federated services containers.

5.6.5 SOA standards

Services Oriented Architecture provides a very extensive set of standards including standards for contract specification, protocols, process orchestration, security, governance and so on. These standards are largely certified by the W3C and OASIS.

5.6.5.1 SOA base standards

The base standards of SOA include

- **WSDL**, the *Web Services Description Language* for specifying services contracts including quality requirements. It is provided in two different levels of abstraction with an *abstract service definition* defining the web-service API and a *concrete service binding* specifying service location, protocol specification, message exchange patterns and data types in the form of XML schemas.
- **WS-Policy** used to specify metadata for services. This standard is used to specify quality requirements like reliability, performance and security requirements.
- **XSD, RDF/OWL, Relax NG** which are all standards for data structure specification.
• **UDDI**, the *Universal Description, Discovery and Integration* which is a community standard for service registry and lookup.

• **WS-ReliableMessaging** which concerns itself with *Quality Of Service* (QOS). one can use this standard to request at least once, at most once, ordered delivery of a message.

### 5.6.5.2 Coordination standards

SOA provides a number of specifications for web-services coordination:

• **WS-AtomicTransaction** provides the standard ACID compliant transaction support for short-running processes,

• **WS-BusinessActivity** is used to assemble a process from atomic transactions which take the process form one consistent state to another,

• **WS-BPEL** is a widely used standard for process orchestration. It supports the specification of sequences, decision points, concurrencies, synchronization points, service request, message correlation, and so on.

• **WS-CDL** is the *Choreography Definition Language* which provides a more flexible way of specifying processes with reduced coupling and without centralized control. One specifies interactions between roles with each party remaining autonomous and a willing participant within an interaction. For example, one can specify a seller-buyer interaction/choreography and parties can choose to play these roles.

### 5.6.5.3 SOA security standards

SOA specifies a range of standards for security including

• the *eXtensible Key Management* (XKMS) which is a standard for key-based authentication

• the *eXtensible Access Control Markup Language* (XACML) for node access control

• the *eXtensible Rights Markup Language* (XrML) for authorization

• *XML-Encryption SSL* for Secure-Sockets-Layer based encryption

• *XML-Digital Signatures* for message integrity.

### 5.6.6 Application concepts and constraints introduced by SOA

SOA applications are assembled within a pipes and filters architecture using stateless services which are orchestrated into higher level languages using an orchestration or choreography language. Special services include transformation elements and filters.

Patterns and constraints for application development include

• *Pipes and Filters* pattern

• *Implement services contract* published in WSDL.

• *Discoverable* on contract or via semantic searches.
5.6. SERVICES-ORIENTED ARCHITECTURES (SOA)

Figure 5.6: SOA application concepts.

- No state is maintained across service requests
- Services must be self-contained in the sense that they may not have any direct dependencies on prior services.
- Services must be self-healing such that failure in realizing one request does not impact subsequent requests.

5.6.7 Levels of SOA use

One commonly makes the transition to SOA and SOE incrementally going from one level of use to the next higher level of use. Typical levels of use are

1. Publish services as web services
   - Services and services contracts published in service registry.
2. Use Services Bus as integration infrastructure
   - Service bus provides routing and adapting.
3. Orchestrate higher level services
   - Preferably in technology-neutral services-oriented process design.
4. Federated services bus
   - Services can be discovered and integrated across services available across federated services bus.

5.6.8 SOA quality attributes

- **Scalability** is typically good through clustering and load balancing.
- **Reliability** is typically good through clustering, compensating transactions and messaging.
- **Flexibility** is also generally very good due to localization of process specification & enforced decoupling via contracts and simplified reuse.
- **Performance** is commonly low because of protocol marshalling, communication and messaging overheads.
- **Auditability** is generally good since it is simple to log all messages which transverse the services bus.
- **Security** is well supported by standards, but remains still quite complex for SOA based architectures.
- **Integrability** is very good with the services bus providing integration infrastructure.
5.7 Space-Based Architecture

On can define a Space-Based Architecture as follows:

**Definition 5.7.1.** The Space-Based Architecture is a reference architecture which specifies an infrastructure where *loosely coupled processing units interact through a shared associative memory* hosting and publishing data objects.

Effectively one can see a Space-Based Architecture as an Event-Based Architecture plus a space.

5.7.1 Aims of SBA

The aims of a Space Based Architecture are to

1. achieve *linear scalability* of stateful, high-performance applications using the tuple space
2. to obtain *increased flexibility* through self-orchestrating processes. There is no central process orchestration – instead processors auto-choreograph processes dynamically. Note also that a single event may result in a tree of processes.
3. to provide an architecture within which difficult problems can be solved.

5.7.2 Overview of SBA

Figure [5.7] depicts the structure of a space based architecture, showing the main components and their responsibilities. Note that the architecture is based, at the first level of granularity, on the *blackboard* architectural pattern.

Figure 5.7: Overview of the space based architecture.
5.7.3 SBA principles

Space-Based Architectures are based on a set of core principles:

- Condensing messaging and data layer into single layer
- Processes via loosely coupled event processors
- High-performance, reliable memory
- Co-locate everything

5.7.4 Elements of a SBA

The core elements of a Space-Based Architecture are

- **Processing units** which contain the processing logic. They have no information about the infrastructure and implement commonly a POJO-based services model.
- The **Virtual middleware** provides the common run-time and clustering model. It is used across the entire middleware stack and includes *data/messaging grid* and *processing grid*.
- An **SLA-based container** enables deployment applications into dynamic pools of machines in such a way such that SLAs (scalability, performance, reliability, ...) are addressed.

5.7.5 Operations on space

The space only supports a set of very simple operations:

- **write**: to put an object into space.
- **read**: to retrieve object from space without removing it — note that the space is a publish-subscribe channel and not a point-to-point channel (a topic, not a queue).
- **take**: to read and remove an object from space.
- **notify**: to register as an observer with space, i.e. processing units register to be notified of certain objects entering space.

5.7.6 Objects, classes and notification

Space-Based Architectures are currently based on an object-oriented approach. You read objects from space and write objects onto space. The objects in space are passive – they are serialized.

The space supports type identification and provides an infrastructure for retrieving the class of an object which is in space. Observation is based on *template object matching* with the observer providing a partially populated object to specify what event they want to receive. Matching is done on

- class type (polymorphically),
- populated fields, and
- non-populated fields seen as wild cards.
5.7.7 An example process in a SBA

Assume we have a knowledge repository where knowledge components mark up information components with semantic relationships like partOf and dependency relationships (see Figure 5.8). The partOf relationship is used to group knowledge components into higher level knowledge domains and the dependency relationship is used to specify that in order to understand knowledge component $a$ you need to first understand knowledge component $b$.

![Figure 5.8: Example semantic relationships used to make up information as knowledge.](image)

The system needs to provide service to generate pedagogically sound training documents from the knowledge repository. The request contains the prerequisites (current knowledge) and the outcomes (required knowledge). The output should be a pedagogically sound document which takes a learner who satisfies the prerequisites to a state where he or she understands the outcomes. The document covers all outcomes and their components and recursively all their dependencies and their components (excluding the prerequisites, their components and their dependencies).

The resultant set of knowledge component needs to be pedagogically ordered such that a when knowledge component is reached, all dependencies have already been covered. Finally, the document needs to be structured into levels of sections (e.g. subsections, sections, chapters, and parts).

5.7.7.1 The document generation process

The process could be auto-choreographed as follows:

1. Lecturer puts document request into space.
2. Knowledge assembler assembles knowledge components to be included.
3. Pedagogical sequencer order knowledge components i.t.o. prerequisites.
4. Document structurer picks up sequenced document and structures it by grouping into higher and higher level sections.
6. Document assembler
   - puts transformation requests to normalized format into space.
   - assembles normalized information components into single document.
7. Document renderer renders document into required output format (e.g. pdf, web, ...).
8. Lecturer is notified of rendered document and picks it up.
5.7.8 Quality attributes

The quality attributes of a space based architecture are typically as follows:

- **Scalability**: Near linear scalability, depending on the implementation of space.
- **Reliability**: Very good based on redundancy of processing units and implementation of space.
- **Flexibility**: Immediate time to market through auto-choreographing processes.
- **Performance**: Depends on communication infrastructure, execution containers, and implementation of space.
- **Auditability**: Everything observable/auditable in space.
- **Security**: Need to enforce secure access to space (authentication/authorization), fine-grained authorization model, and implementation of secure communication.
- **Integrability**: Excellent through space as integration infrastructure.

5.7.9 SBA frameworks

A number of frameworks have been developed which are based on the SBA reference architecture. Some of the more well known are listed below:

- The **Linda Coordination Language** was developed at Yale in early 1980’s. It has an implementation of tuple spaces with space operations. The Linda Coordination Language developed into **Lime** which provides tuple spaces for mobile ad hoc networks, sensor networks, . . .
- **Apache River** (previously Jini) provides the basis for many Java-based SBA implementations.
- **Rinda** is a Ruby implementation of tuple spaces.
- **Blitz** is a JavaSpaces implementation which simplifies configuration.
- **Gigaspace**s and **Tibco ActiveSpaces** are commercial implementations of space-based architecture.

5.8 The AUTOSAR reference architecture

The **AUTomotive Open System ARchitecture** (AUTOSAR) is an open, standardized automotive software architecture, i.e. a reference architecture for automotive control and monitoring systems. The aims of AUTOSAR are to

- Facilitate innovative electronic systems that improve performance, safety & environmental friendliness
- Reduce cost & risk of automotive software development.
- Facilitate increased use of commercial off the shelf hardware
AUTOSAR was developed by automobile manufacturers, suppliers, developers and
the servicing industry. There are a number of open-source as well as proprietary
implementations. ArcCore is an open-source framework which has been published
under the Gnu Public License. There are vendor implementations from a range
of vendors including ETAS, Vector Informatik, OpenSynergy, KPIT Cummins and
MECEL.

5.8.1 Requirements which AUTOSAR was meant to implement

The architectural requirements provided here are taken directly from the official re-
quirements specification. Here we present, however, just a subset.

5.8.1.1 Architectural responsibilities

Some of the architectural responsibilities addressed by AUTOSAR are

- Provide access to system resources
  - processing resources (threads),
  - memory
  - communication resources
- Provide integration infrastructure
- Provide a deployment environment for software components
  - Electronic Control Units (ECUs)
- Provide bridge to operating system
- Scheduling
- Diagnosis

5.8.1.2 Quality requirements

The quality requirements for AUTOSAR include

- Reliability:
  - Redundancy activation,
  - Using AUTOSAR system reliability with failure probability < $10^{-8}$/hour
  - Software components must be protected from each other.
  - Application logic is encapsulated from infrastructure
  - Support SIL-Level-3 development (failure on demand < $10^{-3}$)
  - Possible to check AUTOSAR conformance for implementing frameworks
- Maintainability:
  - Facilitate exchange & update of software & hardware over service life of
    vehicle
  - Releases of AUTOSAR must be forward and backward compatible
5.8. THE AUTOSAR REFERENCE ARCHITECTURE

- **Integrability:**
  - definition of standard interfaces for all architectural components
  - provide open, standardized interfaces for intra- and inter-ECU communication

- **Monitorability/Auditability:**
  - FMEA (Failure Mode and Effects Analysis) compatibility

- **Performance:**
  - minimal performance impacts when used in today’s micro controllers
  - protection of timing requirements is provided by architecture

- **Scalability:**
  - across different vehicle and platform variants
  - Minimal performance impacts when used in today’s micro controllers

- **Portability:**
  - across operating systems
  - across implementing frameworks and vehicles
  - across microcontrollers

- **Security:**
  - software components are protected from illegal access

5.8.1.3 Integration requirements

The integration requirements for AUTOSAR include requirements to integrate with Microcontrollers, Electronic Control Units (ECUs) (sensors and actuators), Analog/Digital converters and Pulse-Width Modulation Devices (PWDs). In addition, AUTOSAR must support pluggable communication protocols including XCP, FlexRay, LIN (Local Interconnect Network) TCP/IP.
5.8.1.4 Architectural constraints

5.8.2 Overview

![Overview of the AUTOSAR reference architecture.](image)

Figure 5.9: Overview of the AUTOSAR reference architecture.

5.8.3 Structural patterns

At the first level of granularity AUTOSAR we look at the structure of AUTOSAR as a whole. At this level of granularity, layering is used with the following layers:

1. Application layer
   - Hosts application components
   - Provides standard interface for application components.
2. A Runtime Environment Layer providing communication services to application software
   - Communication services to application software.
   - Acts as broker and facade, decoupling application components.
3. A Basic Software Layer
   - Provides infrastructural services (persistence, thread access, ...)
   - Decoupled access to microcontrollers, ECUs, ...
   - Device drivers
4. A Microcontroller Layer

The second level granularity component include the Runtime Environment Layer and the Basic Software Layer

The Runtime Environment Layer is based on microkernel pattern:
• Communication bus = AUTOSAR Virtual Function Bus
• routing and adaption between
  – different application components
  – application and infrastructural components

The Basic Software Layer is, once again, based on layering with the following layers:
1. Services Layer
2. ECU Abstraction Layer
3. Microkernel abstraction layer

Other patterns like Bridge and Broker are used at lower levels of granularity.

5.8.4 Architectural tactics specified by AUTOSAR

5.8.4.1 Reliability tactics

AUTOSAR specifies a range of reliability tactics including

• multicast communication to prevent single points of failure,
• redundancy of all architectural elements including redundant drivers, redundant system services, dual microcontroller configurations, multi-channel ECUs, communication redundancy (double I/O paths, redundant data paths, . . . ), as well as application component redundancy.
• Enforcing a defensive approach via fault propagation prevention, terminating flows which encounter faults.
• Event queueing.
• Network & component life cycle synchronization.
• End-to-end communication protection
  – including error detection and correction protocols,
• Requiring explicit error handling within the architecture with fault handling through fault models which may specify removal of service for faulty components, automatically shutting down, or restarting faulty components.
• Correction codes for data and communication.

5.8.4.2 Scalability tactics

Though there is only a single vehicle to monitor and control, the amount of data to be processed across the various sensors and actuators can be significant. To this end AUTOSAR specifies a range of scalability tactics including

• using concurrencies,
• support for scheduling including prioritized scheduling,
• event queueing,
• scaling out resources by clustering.
• signal routing where not an entire message but only the signal needed by the recipient is routed,
• task filtering — executing only tasks which do not have to wait for resources,
• load balancing across embedded multi-core processors,
• multiplexing (carrying many analog signals across a single wire) for network scalability,
• resource management including resource allocation and demand management,
• message aggregation for improved communication scalability.

5.8.4.3 Auditability and monitorability tactics specified by AUTOSAR

Auditability and monitorability are important for vehicular software. AUTOSAR specifies a range of tactics to address these quality requirements including

• built-in support for logging and tracing,
• standard built-in diagnostic, monitoring and debugging services (on-board diagnostics),
• monitoring channels,
• standard interfaces for component testing, and
• built-in testing of architectural services, e.g. RAM and Flash memory testing,

• run-time fault detection — similar to assertions,
• time-out detection,
• data and message corruption detection,
• built-in support for logical and temporal program flow monitoring,
• support for multi-channel ECUs with diagnostic channel.

5.8.4.4 Integrability and accessibility tactics specified by AUTOSAR

The integrability and accessibility tactics specified by AUTOSAR include

• services bus,
• support for an adapter layer,
• request brokering hiding protocol and connectivity details from application components,
• node capability descriptions (similar to IDLs stored in CORBA interface repositories or WSDL contracts sourced)

5.8.4.5 Security tactics specified by AUTOSAR

The security tactics specified by AUTOSAR include

• privilege levels and access control (ECUs),
• memory partitioning to protect components from each other,
5.8.4.6 Integrability tactics specified by AUTOSAR

Even though reliability and monitorability are primary quality requirements, flexibility is still important in the context of upgrading or renewing components which may require a new version of the software. Flexibility tactics specified by AUTOSAR include:

- support for pluggable components through contracts based decoupling and component life cycle management e.g. sleep and wake-up services
- location neutral brokering and support for relocatable components,
- decoupling through a message bus,
- XML-based configuration files,

5.8.5 Integration patterns specified by AUTOSAR

The integration patterns used by AUTOSAR include:

- **Point-to-point channels** (queues)
- **Data Type channels** for channels dedicated to specific message types,
- **Communication bus** (the VFB), **adapters** onto the VFB, and **message brokers**,
- **Publish-subscribe channels** for event subscription,
- **Message aggregator** and later splitting to reduce communication overheads,
- **Message bridge** to bridge across different communication networks used in a vehicle,
- **ContentFilter** for signal routing.
- **Service activators**,
- various message/signal **routers**.

5.8.6 Concepts and constraints for application components

AUTOSAR introduces the concept of software components (SWCs) for application components providing system functionality. They must implement defined APIs and must specify the communication ports through which they are accessed.

The main application components are Electronic Control Units (ECU) which special type of SWCs which monitor and control components in the car, e.g. engine or brake control units.

SWCs/ECUs are constrained to communicate via interfaces in a client/server configuration, and/or via ports in a sender/receiver configuration. In either case all communication is routed through the the RTE (Runtime Environment) represented by VFB (Virtual Function Bus).

AUTOSAR specifies structures for SWC/ECU resource descriptors (XML). The descriptors are used to specify:

- the sensors and actuators controlled and/or monitored by the ECU,
• the memory requirements for the ECU,
• the processor requirements for the ECU, and
• the required communication periphery.

5.8.7 Trade-off decisions made in AUTOSAR

In AUTOSAR development and infrastructure cost is traded off for reliability, monitorability and auditability. Furthermore some performance is traded off for flexibility, in particular for decoupling, pluggability and portability.

5.9 Cloud based architectures

A cloud-based architecture is an architecture where computing resources are provided over the network. The history of cloud-based architectures can be traced back to the work done by John McCarthy in 1960 around computing as a public utility. Early open-source frameworks for cloud computing include Eucalyptus and OpenNebula, both of which were released in 2008.

5.9.1 Overview

![Diagram of Cloud-Based Architectures](image)

Figure 5.10: Overview of Cloud-Based Architectures.

5.9.2 Types of cloud services

Cloud Base Architectures provide different types of services which are categorized at the high level as follows:
• **Infrastructure as a service (IaaS)**  
  − physical/virtual machine, networking, disk space, . . .

• **Platform as a service (PaaS)**  
  − Operating system, databases, application server, ESB, . . .

• **Software as a service (SaaS)**  
  − per usage charging (e.g. accounting services, email services, ERP as a service)

• **Storage as a service (STaaS)**  
  − Rent space on storage infrastructure (relational or noSQL)

• **Desktop as a service (DaaS)**  
  − Desktop virtualization

• **Security as a service (SECaaS)**  
  − authentication, anti-virus, anti-malware/spyware, intrusion detection, security event management, . . .

• **Test environment as a service (TEaaS)**  
  − on-demand test environment

• **API as a service (APIaaS)**  
  − Multiple entry points for API calls  
  − e.g. REST, XML web services or TCP/IP.
Chapter 6

Architecture design methodologies

6.1 Attribute Driven Design (ADD)

Attribute Driven Design (ADD) is a high-level architecture design method which aims to guide experienced architects in designing a conceptual software architecture. It was developed around 2005 by Woyceck, Bachmann, Bass and Clements [?, ?] of the Software Engineering Institute (SEI) at Carnegie Melon.

The aim of ADD is to define architecture design process which is based on quality attribute requirements, and which decomposes system applying architectural tactics and patterns across components. The focus is on addressing non-functional requirements but using ADD, the software architecture still partially determined by functional requirements.

6.1.1 Inputs and outputs of ADD

In ADD the software architecture requirements include

- Functional requirements
  - as list of use cases.
- Quality requirements.
  - Scalability, reliability, performance, security, auditability, integrability, portability, modifiability, cost, ...
  - Must be prioritized and quantified.
- Architectural constraints
  - e.g. technology constraints or hardware constraints.

Note that there are

- no architectural responsibilities, but instead there are functional requirements,
- no requirements across levels of granularity,
no integration requirements except perhaps artificially as architectural constraints.

The ADD output is a software architecture specification which includes:

- roles and responsibilities,
- relationships,
- Properties (quality attributes).

These are specified using combination of architectural views. The views are, however, not precisely specified. The authors suggest a module view, a component-connector view, and a hardware allocation view.

Some concerns one could raise against ADD include that:

- there is no separation between application and architectural components.
- that the specification structure is not specified (i.e. there is no metamodel),
- it is not clear how to specify tactics and how these are concretely realized.
- the traceability between requirements and design not clear, and
- that the software architecture specification does not include concepts and constraints for application components.

### 6.1.2 Steps of the ADD method

Figure 6.2 shows an overview of the *Attribute-Driven Design* process.

During the first step, the *assess requirements* step one confirms that one has sufficient requirements, i.e. that the requirements are quantified and prioritized, and that they are linked to business requirements and mission goals.

Each quality requirement is specified in a *stimulus-response* form which is based on quality attribute scenarios. For each stimulus response one specifies:

1. the stimulus source (e.g. users),
2. the stimulus (e.g. initiate transaction),
3. the artifact (e.g. system),
4. the response (e.g. transaction processed), and
5. a response measure (e.g. latency of below 1 second)
In ADD, the stimulus response specification guides architecture design, verification and quality attribute testing.

During the component selection phase one selects the component to decompose. This step fixes the context for the current level of granularity. At the first level of granularity the selected component is the system as a whole.

For lower levels of granularity one has a partitioning of the current level of granularity into components. One selects one of those components as the context component for the next lower level of granularity. For this one can be guided by

- the current knowledge of architecture,
- risk and difficulty concerns,
- business criteria (e.g. impact, skills availability, ...), and
- dependencies of other components.

### 6.1.2.1 Identify candidate architectural drivers

During the identify candidate architectural drivers step, one performs a priority \(\Rightarrow\) impact mapping resulting in a fuzzy impact assessment represented by (H,H), (H,M), (H,L), (M,H), ..., (L,L). In ADD one is required to select several (they suggest top 5 or 6) which become the candidate architectural drivers. The focus requirements for subsequent steps in architecture design process is then on these candidate architectural drivers. Note, however, that further analysis may change selection somewhat, i.e. one may find that certain requirements have more impact than initially thought.

During the 4th step of ADD, one chooses appropriate design concept. Note that the authors see tactics as types of patterns. The steps in this phase are

1. Identify architectural concerns associated with candidate drivers
   - e.g. fault prevention, detection, recovery for reliability.
2. Create list of patterns that address concerns.
   - e.g. resource reuse, load balancing, ..., for scalability
3. Select patterns.
   - Per driver specify pros & cons for each pattern.
• Record rationale for selection.

4. See how patterns relate with one another
   • Combine patterns for overlapping functionality.
   • Results in higher level pattern (according to authors).

5. Describe patterns using views.
6. Evaluate and resolve inconsistencies.
   • Evaluate against requirements.
   • Check for conflicts.
   • Effectively here they make trade-off decisions.

Step 5 in ADD is the instantiate elements and allocate responsibilities step. It is decomposed into a number of sub-steps:

1. Specify components for all patterns identified in step 4.
   • These components are the child components of the context component.
2. Assign architectural responsibilities to all child components.
   • To realize tactics.
3. Allocate functional responsibilities assigned to parent across child components.
   • e.g. banking system must support cash collection, recording of transactions, ...
   • Assign these responsibilities to child components.
   • Consolidate components with similar responsibilities.
4. Create additional components
   • If similar functional responsibilities allocated to same component, but have different quality requirements.
   • To separate responsibilities (improve modifiability & pluggability).
5. Analyze and document design decisions using views
   • Module views
   • Component-connector views
   • Allocation views (how software components allocated to hardware components).
6. Refine with details
   • Resource requirements for components.
   • Integration technologies, protocols, ..., and data models used.
   • Computational elements and process/thread models
   • Scheduling strategies.
   • Execution and activation/deactivation dependencies.
   • ...

Step 6 of ADD is the define component interfaces step. For each component one specifies an interface with

1. operation/service signature,
6.2. SYSTEMATIC METHOD FOR ARCHITECTURE DESIGN (SYMAD)

2. specification of inputs and outputs,
3. pre- and post-conditions,
4. quality requirements.

During the last step, the verify and refine step, one verifies that the component satisfies

1. functional requirements,
2. quality attribute requirements, and
3. architectural (design) constraints.

6.1.3 ADD as an iterative process

ADD is as an iterative process which is repeated across components within levels of granularity and across levels of granularity.

6.2 Systematic Method for Architecture Design (SyMAD)

6.2.1 The bigger picture

Figure 6.3: SyMAD in the context of model driven development.
6.2.2 Overview of SyMAD

Figure 6.4: An overview of the SyMAD method showing method steps and reuse exploration.

6.2.3 Case study: A simplified banking system

As a case study we consider the design of a software architecture for a simplified banking system. The case study illustrates

- the identification of architectural responsibilities and the allocation of these responsibilities to architectural components,
- the reuse of architectural components,
- the selection of architectural patterns and strategies,
- how one introduces concepts and constraints for application components,
- the traversal across levels of granularity and the architectural design of lower level components, and
- the interplay between software architecture and application design.
6.2.3.1 System overview

The case study subject is a regular banking system used by individual and corporate clients for transactional banking, loans and deposits. The banking services need thus be remotely accessible by humans and client systems.

In addition to standard banking processes, the system must facilitate the maintenance of customer information and the generation of regulatory reports which need to be fed through to the industry regulator.

6.2.3.2 First level of granularity

The software architecture for the system as a whole is selected as the context component for which we need to elicit the architectural requirements, check for reuse and potentially design the software architecture.

6.2.3.2.1 Software architecture requirements  The requirements for the first level of granularity include the specification of the high-level architectural responsibilities, the quality requirements, and the architectural constraints for the banking system.

6.2.3.2.1.1 Architectural responsibilities  The first level granularity architectural responsibilities identified from the requirements are shown in Figure 6.5. It includes the responsibilities of providing various access and integration channels as well as the responsibilities of hosting business processes and data, and that of providing reporting services. Note that these are all infrastructural responsibilities which are not specific to the actual banking functionality.

Figure 6.5: Architectural responsibilities for the first level of granularity.

6.2.3.2.1.2 Quality requirements  Figure 6.6 shows the quality requirements for the system as a whole. Security, scalability, reliability and auditability are regarded by the client as the most important quality attributes followed by performance and maintainability.

6.2.3.2.1.3 Architectural constraints  Assume that the client requires that no third-party elements of the system should be vendor locked, i.e. all elements should be either standards compliant and hence pluggable or available from the open source community.
6.2.3.2.2 Check for reuse In SyMAD one already checks for reuse at the first level of granularity.

6.2.3.2.2.1 Identifying candidate reference architectures and frameworks
The system is an enterprise system with typical architectural responsibilities and quality requirements which are similar to those of many enterprise systems. There are a range of reference architectures and frameworks targeting such systems. Examples of reference architectures include Java-EE [?] and SOA [?]. For both of these there are a range of implementing frameworks (e.g. RedHat WildFly, IBM Websphere and Apache ServiceMix). In addition there are a range of frameworks targeting enterprise systems which are not based on a reference architecture. Examples include the Spring Framework and Microsoft.Net [?].

In order to reduce the bulk of the analysis we will confine ourselves to the comparative assessment of only three of the above, Java-EE, Microsoft.Net and SOA. We need to assess these candidate reference architectures and frameworks for the extent to which they are able to satisfy the architectural requirements for our banking system. In particular, only reference architectures and frameworks within which it is possible to meet the quality requirements and the architectural constraints can be considered. In addition one needs to assess to which extent the architectural responsibilities are addressed and how one would plug in additional architectural components in order to address any architectural responsibilities not addressed by the architectural component.

6.2.3.2.2.2 Assess against constraints Both reference architectures comply to the architectural constraint that the architectural elements should not be vendor locked. The two reference architectures are, additionally, standards compliant.
with the standards being community managed standards. Hence, by considering the architectural constraints we exclude only Microsoft.Net.

6.2.3.2.2.3 Assess alignment of quality requirements We first compare from a quality attributes perspective the two reference architectures, SOA and Java-EE. In SOA, the dominant quality attribute is integrability and flexibility (time to market) with the reference architecture designed around the microkernel pattern with the services/integration bus being the central focus of the reference architecture. Integrability is further improved by using a standard text-based protocol (SOAP/HTTP) and having services contracts published in service registries in a standard way. Flexibility and “time-to-market” are good with higher level services being easily “orchestrated” from published lower level services sourced across systems connected to the services bus. Services are stateless and process state is maintained within the messages which are transferred across all services employed in the process specification. Queueing is used to improve reliability. Auditability is provided through the ability of logging all messages communicated over the services bus. Even though SOA-frameworks use thread pooling and clustering to improve scalability, the overheads introduced by the above architectural elements result in performance and scalability being traded off for integrability, flexibility and time-to-market.

Java-EE, on the other hand, is based on the layered architectural pattern component model and is used with stateless and stateful application components. Communication is done either via native method calls or via an efficient binary protocol (RMI/CORBA/TCP/IP). Component contracts are specified in CORBA’s Interface Definition Language (IDL). Process state is commonly maintained within the session and not transferred with messages exchanged between components. The reference architecture employs a range of tactics to address scalability including general resource pooling—such as thread, object, and connection pooling—, database caching, and clustering. Integrability is supported through supporting CORBA and Web Services standards and through providing a connector architecture within which connectors for non-standards based integration are developed. The reference architecture has good security support with authentication, encryption and role-based authorization at the business logic layer all done through interception. Reliability is largely addressed through clustering, transactions, and support for messaging. Auditability can be implemented through global interceptors which log all exchanged messages. The dominant quality attributes are scalability and security. Flexibility and integrability have, to some extent, been traded off for these.

Referring back to the quality requirements specified in Section 6.2.3.2.1.2 we note that Java-EE provides a better fit than SOA. The scalability and performance requirements can be shown to be readily achievable (this can be further confirmed with a proof-of-concept implementation of the software architecture). The reference architecture makes provision for all the security requirements and reliability is achievable in the context of making use of clustering as well as setting up a disaster-recovery site. Request and response logging can be done via a global logging interceptor. Entity modification logging is slightly more involved but can be done within Java-EE using a JPA (Java Persistence API) interceptor. Finally, since the Java-EE reference architecture is an industry standard with well over 20 framework implementations provided by a range of vendors and open source projects with heavy industry investment in these standards, it can be reasonably be expected to be maintainable
over the next 20 years. The wide skills availability for this reference architecture also reduces maintainability risk.

The scalability, reliability and security requirements would be achievable in SOA, but the complexity and associated cost would be higher than in Java-EE. Message logging is straightforward in SOA, but entity change logging is not supported and would have to be provided by additional persistence infrastructure. Performance would be a major risk factor when choosing SOA as the reference architecture. Furthermore, the standards compliance is less mature within SOA frameworks. As a result, applications developed within this reference architecture are commonly more framework locked than applications developed for the Java-EE reference architecture—this results in a higher maintainability risk.

Figure 6.7: Architectural components required for architectural responsibilities.

6.2.3.2.4 Assess addressing of architectural responsibilities In order to address the architectural responsibilities specified in the requirements (see Section 6.2.3.2.1.1), we require that the chosen reference architecture must provide architectural components addressing these responsibilities. Figure 6.7 depicts the required architectural components. The web broker needs to provide web-page access, generates the service requests from the provided information and renders the return values back onto the web. Asynchronous system integration to humans is usually done via emails. An email adapter needs to render requests and responses originating from the banking system to an email and, for a request, it needs to provide a web link for humans to provide their responses. A web-services broker maps SOAP-based web services requests onto actual method calls and asynchronous system access is done through message queues. The remainder of the required components shown in Figure 6.7 are self-explanatory.

We need to assess now to what extent the two candidate reference architectures provide specifications for the required components. In the case of SOA, both web services and messaging brokers are provided, but human access needs to be addressed through other components which would be plugged into the reference architecture (e.g. using a JSF-supporting web container or a Django front-end). SOA also has good support for business process execution engines which are plugged into the ESB—the most commonly used engines are BPEL and jBPM engines. However, even though some work has been done on providing SOA with a persistence infrastructure [?], no standard is thus far widely adopted or supported.
Java-EE, on the other hand, does provide specifications for most of the required components and implementing frameworks provide implementations for these specifications. Figure 6.8 shows the mapping of the abstract components onto the components as specified in the Java-EE reference architecture as well as the components which are not provided by the reference architecture. These components are selected for the next lower level of granularity where the requirements specification for them is done. We will then either source existing architectural components which meet these requirements or design appropriate architectural components. The diagram pre-empts the results from the next level of granularity by showing which components can be sourced and which need to be designed.

6.2.3.2.5 Component selection Both SOA and Java-EE satisfy the architectural constraints, but the quality attribute trade-offs made in Java-EE are better aligned with the quality requirements for the banking system, with Java-EE having advantages on scalability, performance, security and maintainability. SOA is stronger when it comes to integrability, but this is not a central quality requirement for this case study. Furthermore, Java-EE addresses more of the architectural responsibilities. The analysis thus leads us to select the Java-EE reference architecture for the system as a whole.

We could go a step further and also select a particular implementing framework (e.g. RedHat WildFly or IBM Websphere Application Server).

6.2.3.2.3 Configuring the architectural component We have chosen the Java-EE reference architecture for the architectural component of the first level of granularity providing the infrastructure within which the non-functional requirements are addressed and within which application logic will be deployed and executed. The reference architecture provides some freedom around how architectural responsibilities are addressed, how the infrastructure is configured, which architectural tactics are used to address quality requirements, and which concepts and constraints are introduced for application development. These are specified when configuring the reference architecture.
When configuring the re-used architectural component at the current level of granularity (in this case study, the Java-EE reference architecture), we need to map the architectural responsibilities onto components of the re-used reference architecture and configure the infrastructure, architectural tactics and application concepts and constraints of the re-used component.

6.2.3.2.3.1 Configure architectural tactics

In our chosen reference architecture, some architectural tactics are always used, whilst others are optional. For example, resource pooling (including thread, object and connection pooling) is always applied in Java-EE. However, to address the scalability and reliability requirements, we choose to configure the architecture with clustering and replication to a disaster recovery site.

As mentioned previously, these tactics are often cross cutting concerns which are applied across a subset of the architectural components. Hence they are naturally modeled as aspects. For this reason we choose the AO-ADL to document the application of architectural tactics. In addition to providing connections between architectural components, Figure 6.9 shows the application of tactics to the system as a whole. In AO-ADL aspectual roles represented by solid arcs to connectors represented by largish circles (e.g. C1).

6.2.3.2.3.2 Mapping architectural responsibilities onto architectural components

During the process of assessing the Java-EE reference architecture against the potential alternatives, we already performed the mapping of architectural responsibilities onto the components specified in the Java-EE reference architecture (see Figure 6.8).

6.2.3.2.3.3 Configuring the infrastructure

Being a reference architecture, Java-EE provides a template software architecture which needs to be configured. The Java-EE configuration for this level of granularity is shown in Figure 6.10.
For our designed software architecture we choose a JSF/Facelets based presentation layer (instead of Servlets/JSP). Furthermore, we constrain message driven beans to be purely a messaging adapter to business logic contained in stateless sessions beans – hence, in our designed architecture the message driven bean based router is part of the access layer, even though it is technically deployed within the EJB application server.

Persistence is done through JPA based persistence (i.e. using an object-relational mapper, object cache and queries and relationships specified within object graphs and not tables). The Email adapter, JCA-based regulator adapter and JPA-based persistence adapters are all part of an infrastructure layer.

**6.2.3.2.3.4 Configuring application concepts and constraints** For application logic components Java-EE provides the concepts of stateful and stateless session beans as well as message-driven beans and entities. To simplify the banking system, facilitate more efficient passivation and thread sharing, and improve reusability we restrict all application functionality to be specified within stateless session beans and all domain objects within entities. Message-driven beans are specified to be used solely for messaging adapters, i.e. demarshall messages received from incoming queues or topics, map them onto requests made to stateless session beans and marshall the return values received from these beans back onto a message which is put onto a response queue.


6.2.3.3 Lower levels of granularity

Some of the architectural components are not specified by Java-EE. In particular, the database, reporting engine and email adapter are not specified by the reference architecture and also not provided by Java-EE based frameworks. We assigned these responsibilities to lower level architectural components, but the requirements, reuse assessment and design of these components is done as we traverse to lower levels of granularity.

Some of the lower level architectural components may be sourced (re-used) and need not be designed. However, we still need to still base the reuse selection on architectural requirements for that component. For example, it is unlikely that one will design a new database for a typical enterprise system. However, the choice on the persistence architecture used will depend on, for example, the performance, scalability, flexibility and integrability requirements. Common choices include relational databases, key-value stores, document stores, graph/object stores, ... Concrete designs and realizations of these stores employ different tactics to achieve quality requirements like scalability (e.g. eventual consistency). For a banking system it is likely that integrability requirements will result in a selection of a relational database management system like PostgreSQL or Oracle RDBMS. We thus select/reuse a relational database as architectural component to address the persistence responsibility and do not need to design that component.

Assume that there are, however, no standard email adapters which map service requests which happen to be made to a human being onto an email with a link to a web page which captures the response message. This component would have to be architected and its functionality needs to be designed.

6.2.3.4 Reporting Engine

The second architectural component of the banking system which is not specified within the Java-EE reference architecture is the reporting engine. We would specify the architectural requirements for the reporting engine and then check for re-use. It is likely that one of the standard reporting engines (e.g. JasperReports or Pentaho [?]) would fulfill the requirements for the reporting engine. The chosen engine would have to be configured to meet the architecture requirements of the banking system. This process would be similar to that of configuring the Java-EE reference architecture.

6.2.3.5 Email Adapter

The system submits asynchronous messages and requests to external parties in a way which is independent of whether the messages are processed by external systems or humans. The responsibility of the email adapter is to map asynchronous messages and requests onto an email for asynchronous human consumption and, in the case of requests, provide a response capture facility which enables humans to submit a response through a web page whose content is demarshalled, with the demarshalled message being provided asynchronously back to the system.

From the perspective of the banking system, the email adapter is purely an infrastructural/architectural component—it is just an adapter and does not provide any application functionality for banking. However, at this lower level of granularity we
have both non-functional and functional requirements and hence will have to do both architecture and application design for this component. Without going into much detail on the side of application design, this example also illustrates how both software architecture and application design are done across levels of granularity.

6.2.3.5.1 Architecture design for the email adapter
The quality attributes of a lower level component need to support the qualities of the architecture at the higher level. Furthermore, the lower level architectural components (e.g. the email adapter) are deployed within the architecture of the parent component (the banking system). For example, the processes of the email adapter will be executed within the process execution engine (the EJB container) of the parent component. In this context, lower level components benefit from the architectural tactics which have been applied to the parent component (e.g. resource pooling in the context of thread and connection pooling).

6.2.3.5.1.1 Software architecture requirements for the email adapter
The software architecture requirements for the lower level component are constrained by the quality requirements and architectural constraints of the parent component. Figure 6.11 depicts the architectural responsibilities which need to be addressed by the infrastructure within which the application logic for the email adapter is to be developed.

The architecture of the email adapter needs to provide the infrastructure to integrate with the banking system’s asynchronous input and output channels (the message queues) on the one side and the mail server and human beings on the other side. In particular, it delivers asynchronous messages to humans via email and captures the asynchronous responses from humans via the web. In addition, the email adapter also requires an execution environment within which its processes can be executed.

The quality requirements for the email adapter are directly derived from the quality requirements of the higher level component – the banking system as a whole. To do this each higher level quality requirement was analyzed for its relevance at this lower level of granularity. Those quality requirements which were relevant were quantified in such a way that the quality requirements for the email adapter support those for the banking system as a whole. The resultant quality requirements for the email adapter are shown in Figure 6.12.

The software architecture of the lower level component must be designed within the architectural constraints of the parent component. Hence no third-party elements of the email adapter may be vendor locked.

6.2.3.5.1.2 Check for reuse
Assume we cannot source an architectural component satisfying the requirements for an email adapter. Of course, we may still reuse
lower level components within the email adapter—the opportunity for such reuse will be investigated as we traverse to lower levels of granularity.

**6.2.3.5.1.3 Architecture Design** We thus enter the software architecture design phase illustrated in Figure ?? . We start with selecting architectural tactics through which we address the quality requirements for the email adapter. We then allocate the architectural responsibilities identified during the requirements phase to architectural components of the email adapter before designing the infrastructure between these components. Finally we specify the concepts and constraints within which the application logic for the email adapter is to be specified.

Architectural tactics are selected to concretely address the quality requirements specified previously. Table ?? can be used to assist with the selection of appropriate tactics, i.e. tactics which strengthen the sought quality attribute without having too negative an impact on other required quality attributes.

*Confidentiality* will be achieved through encryption of both the email-based requests and the HTTP responses (using HTTPs).

Some *scalability* tactics are inherited from the system as a whole, i.e. the email adapter already achieves certain levels of scalability through the *clustering* and *load balancing* applied to the banking system as a whole. In addition to those tactics, the email adapter will use *thread and connection pooling* as managed through the higher level EJB-container as well as *caching* of the response capture pages as done through the web container. Finally we also use *queueing* of requests to be sent by the adapter and responses captured by the adapter to further spread load over time and hence improve scalability.

In order to address *Reliability*, we choose *persistent queueing* of both request and response messages with message removal from queue not at message delivery, but after acknowledgement of successful processing of the message. The reliability of the email adapter also benefits from the reliability tactics introduced for the system as a whole, including the clustering, replication, and the usage of a disaster recovery site.

Finally, to address the *auditability* requirements we perform logging of all messages as they are put onto the request and response queues as well as all emails sent and all captured user responses.

We need to assign the architectural responsibilities listed in the requirements specification (see Section ??) as well as the responsibilities of realizing the tactics to required architectural components.

Figure ?? shows how the architectural responsibilities are assigned to required architectural components (represented by required interfaces in UML). We also preempt the mapping of those required components onto the aspects of the architecture of the email adapter which are provided by virtue of having chosen the Java-EE reference
Figure 6.13: The architectural tactics applied to the email adapter.

architecture at the previous level of granularity (that mapping would only be done as we go to the next lower level of granularity for the required architectural components). The remaining components will need to either be externally sourced (e.g. an email encryptor) or designed at the lower level of granularity (e.g. the logging interceptor).

The architecture within which the functionality of the email adapter is to be developed is assembled from the architectural components addressing the non-functional requirements. The components implementing architectural tactics can largely be modeled as aspects in the spirit of aspect-oriented software architecture design [?].

6.2.3.5.2 Email adapter application design

The email adapter represents an asynchronous requests adapter to humans, i.e. it receives an asynchronous request which is to be processed by a human, generates a web page through which the response for the request can be captured as well as a response adapter which maps the HTTP post message received through that web page onto the response message which is put on a response queue.

For the application design one can use any architecture- and technology-neutral design method. This will introduce application components for the email adapter which need to be deployed in the software architecture of the email adapter which is, in turn, part of the architecture for the banking system. For this example we only sketch the first level granularity application design of the email adapter using the URDAD method (mentioned in Section ??).

6.2.3.5.2.1 Services Contract

Figure 6.16 depicts the services contract for the first level of granularity for the email adapter.

6.2.3.5.2.2 Functional requirements

In URDAD, functional requirements is taken literally, i.e. as requirements for functions/services. Figure 6.17 depicts the required functions and the allocation of these to services contracts.
6.2.3.5.2.3 Process design During the first-level granularity process design phase for the email adapter we assemble a process across the functions/services specified during the functional requirements phase. Figure 6.18 depicts a process design for the email adapter.

6.2.3.5.3 Lower levels of granularity Both the application and software architecture design for the email adapter will have to be taken through lower levels of granularity until all application and architectural components are assembled from available components. At any level of granularity we may have both software architecture design addressing the non-functional requirements for that level of granularity and application design addressing the functional requirements for that level of granularity.

Figure 6.14: The allocation of architectural responsibilities to required architectural components as well as the mapping onto corresponding components provided by the Java-EE reference architecture.

Figure 6.15: The infrastructure of the email adapter.

Figure 6.16: The services contract for the email adapter.
6.2. SYSTEMATIC METHOD FOR ARCHITECTURE DESIGN (SYMAD)

Figure 6.17: The functional requirements and responsibility allocation for the processRequest use case.

Figure 6.18: The process design for the email adapter.
Chapter 7

Software architecture description

Currently we do not have a standard for architecture descriptions. We do, however, have a standard for the requirements of an architectural description and there are some promising attempts to provide us more useful architecture description frameworks and languages.

7.1 Kruchten 4+1 Views

The Kruchten 4+1 approach to architectural description is based on a collection of views which are generally documented using UML diagrams — i.e. the Unified Modeling Language (UML) is used as an Architecture Description Language (ADL). The approach was developed by Philippe Kruchten at Rational Software [?].

The approach is relatively widely adopted in industry. The aim is to be able to create a practical architecture description relatively rapidly, without being caught in analysis paralysis.
7.1.1 Views

Figure 7.1: Kruchten 4+1 Views.

7.1.1.1 Logical view

The Logical View is concerned with the functionality the system provides to end-users, i.e. with specifying application functionality. The UML diagrams used to document this view are class diagrams, communication diagrams and sequence diagrams.

7.1.1.2 Process View

The Process View is used to specify the dynamic aspects of the system. It is meant to describe the system processes and how processes communicate. The focus is on the runtime behavior of the system including the specification of concurrencies. The view is used to describe how some non-functional aspects are addressed, namely performance and scalability. The approach suggests that UML activity diagrams are used to document this view.

7.1.1.3 Development or implementation view

The development view or implementation view describes the software system from a software management perspective. It focuses on the decomposition of the system into software modules and subsystems. The UML diagrams used to document this view are package diagrams and component diagrams.

7.1.1.4 Physical view

The Physical View or Deployment View depicts the software system from an engineer’s point-of-view. It describes the deployment of components onto physical components or frameworks, i.e. the deployment onto nodes. It also depicts the physical connections between the components.
The description of the physical infrastructure may depict some non-functional concerns, like, for example, reliability/availability and scalability.

The UML diagram used for this view is the implementation (deployment) diagram.

### 7.1.1.5 Use-Case View

The Use Case View or Scenarios View depicts the functionality of the software system from user’s or stakeholders perspective. It documents the functional requirements and the use case interactions from external objects perspective.

Commonly the view is restricted to a small set of use case scenarios. It illustrates application processes within architecture. The view is sometimes used to develop “architectural prototype” and also for scenario-based architecture validation.

The UML diagrams used for the use case view include use case diagrams, sequence diagrams, and communication diagrams.

### 7.1.2 Criticisms raised at Kruchten 4+1

The Kruchten 4+1 approach does not define a separation between architecture and application design. Furthermore, there is no defined model structure and hence the framework is not amenable to automated model validation and artifact (e.g. code, test, documentation) generation.

Furthermore, the approach does not have any explicit support for core architectural concepts like structural and integration patterns, architectural tactics and also does not have any explicit way of specifying concepts and constraints for application components.

### 7.2 Architecture Description Languages (ADLs)

An ADL is a language which is meant to use to describe a software architecture. The definition one accepts for software architecture will change what needs to be described and may require that one uses a different ADL. Since we do not have consensus on what the scope of a software architecture is, we do not have a standard for an ADL – in fact well over 50 ADLs have been developed, though none of these has wide-spread acceptance and is in wide practical use.

### 7.2.1 Scope of an architectural description

The description scope of an ADL differs from ADL to ADL. Most ADLs are able to describe components and connectors. ADLs differ on whether they

- whether higher level components can be assembled from lower level components,
- have explicit support for patterns and tactics,
- have the concept of a responsibility,
- they can document functionality and processes, and
• whether they have a way to specify concepts and constraints for application components.

Also, one often would like to document both, a conceptual architecture as well as an implementation architecture which specifies the chosen technologies and frameworks.

7.2.2 Architecture description approaches

There are a number of different approaches taken with software architecture description:

• Multiple independent views like Kruchten 4+1

• A single model which may be supported by diagrammatic and/or text syntaxes.

• Multiple linked models.

• Single model populated through multiple views facilitating model checking. The approach may be supported by formal specification of semantics in the form of an ontology.

• Formal model specifications

7.2.3 Aspect-Oriented ADL

The Aspect-Oriented Architecture Description Language (AO-ADL) was developed by Pinto et al around 2007 [? , ?]. The premise is that non-functional requirements are typically cross-cutting concerns, and hence that they are not suitably represented in traditional component-connector approach.

AO-ADL enriches a traditional component-connector ADL with aspect components implementing architectural tactics. As is standard in an aspect-oriented approach, these aspect components are applied to point cuts – a point cut is a set of join points where the aspect component is applied as an interceptor.
7.2.3.1 AO-ADL MDD process

![Diagram of the AO-ADL MDD process](image)

Figure 7.2: The AO-ADL MDD process.

7.2.3.2 AO-ADL concepts

In AO-ADL components model both non-cross-cutting concerns with required/provided roles semantics (as in UML) as well as cross-cutting concerns. Components which model cross-cutting concerns introduce *aspectual roles*.

The connectors have thus normal connection semantics extended with *aspectual bindings*.
7.2.3.3 AO-ADL metamodel

Figure 7.5: The AO-ADL metamodel.
7.2. ARCHITECTURE DESCRIPTION LANGUAGES (ADLS)

7.2.3.4 Example aspects application

Figure 7.4: AO-ADL’s extensions to standard connector semantics — text syntax.

Figure 7.6: A simple AO-ADL example.
7.3 IEEE/IEC/ISO 42010

IEC/ISO/IEEE 42010 is a standard specifying the requirements for an architectural descriptions of a software-intensive systems. It does not specify architecture description framework, i.e. the structure or content of an architectural description. That is left to description framework (RM-ODP, Kruchten 4+1, . . . )

Instead IEC/ISO/IEEE42010 specifies at a more abstract level that any compliant architecture description documentation should be such that

- it is in the form of multiple views with each view complying to a specification for that type of view,
- the rationale links for any architectural decisions are included in the description, and
- that it is independent of the architectural process used.

7.3.1 History of IEC/ISO/IEEE 42010

Work on a standard for the requirements of an architectural description started in 1995. The first specification, IEEE-1471, was approved in 2000 by the IEEE-SA standards board. In 2007 IEEE/ISO 1471 was adopted as an ISO standard. A rework of the first standard was completed in 2011 resulting in IEC/ISO/IEEE 42010.

7.3.2 What does IEC/ISO/IEEE 42010 provide?

IEC/ISO/IEEE 42010 specifies a metamodel which any architectural description should comply to. It enforces traceability of decisions to stakeholder concerns and that the architectural descriptions is specified through multiple views which comply to view-specifications called view points.

The metamodel provides for capturing the rationale behind architectural decisions, inconsistencies, and unresolved issues.

Furthermore, the standard also specifies requirements for architecture description Frameworks and for Architecture Description Languages (ADLs).

7.3.3 Software architecture definition

The definition of software/system architecture has changed from IEEE/ISO 1471/2000 to ISO/IEC/IEEE 42010:2011. In the earlier IEEE/ISO 1471 specification, software architecture was defined as

**Definition 7.3.1.** Software Architecture is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.

With the 2011 release of ISO/IEC/IEEE 42010 architecture is defined more widely as

**Definition 7.3.2.** The architecture of a system is the fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and the principles of its design and evolution.
7.3.4 Stakeholders and stakeholder concerns

A stakeholder represents a role which has an interest in the architecture. Note that the architecture stakeholders are not simply the stakeholders in the system use cases. For example, an insurance regulator may have an interest in how an insurer processes a claim, but might not have an interest in the architecture used to process the claim. Conversely, the system maintenance team may have an interest in the architecture (e.g. its monitoring support), but might not have an interest in how claims are processed.

7.3.5 Envisaged uses of architectural descriptions

IEC/ISO/IEEE 42010 envisages that architectural descriptions are used

- to provide documentation for an existing or envisaged architecture,
- to provide guidelines for the evolution of the architecture and to document the persistent characteristics, and supporting principles to guide acceptable change.
- to communicate the architecture across stakeholders and to simplify the communication between stakeholders,
- to facilitate the assessment, measurement and comparison of architectures in a consistent manner,
- to assist with planning, managing and executing system development,
- to facilitate verification of a system’s compliance with an architecture description, and
- to record potential contributions to the body of knowledge for software-intensive systems architecture.

7.3.6 Normative elements

The normative or mandatory elements of a IEC/ISO/IEEE 42010 compliant architectural description are

1. Contextual information like issuing and client organizations, date, revision history, reader guidelines, ….
2. An identification of the architecture stakeholders.
3. For each stakeholder, the architectural requirements/concerns.
4. Models and or descriptions of the system organized in views corresponding to defined view points.
5. The view point details explaining the artifacts of the views corresponding to the view points.
6. The rationale used to select the described architecture and evidence that credible alternatives were considered.
7. A list of open issues, uncertainties and potentially inconsistencies across views.
7.3.7 View points and views

Each view communicates some aspect about the architecture of the system. It has to adhere to comply with a view point specification. The IEC/ISO/IEEE 42010 specifies both, mandatory and optional elements for a view point specification.

7.3.7.1 What is a view point?

A view point is defined as follows:

**Definition 7.3.3.** A view point is a perspective from which the architecture is specified/shown.

A view point typically addresses a particular architectural concern like responsibility allocation, persistence, integration, tactics and so on.

The IEC/ISO/IEEE 42010 specification does not specify any view points. This is left to individual architectural frameworks and methodologies. The specification simply specifies that the view points need to be defined.

7.3.7.2 Mandatory elements of a view point specification

The IEC/ISO/IEEE 42010 specification cor the requirements for an architectural description requires that a view point definition includes

- a view point name.
- the source, if any, of the view point (e.g. author, citation, . . . )
- the rationale behind the view point.
- the stakeholder concerns which are addressed through the view point.
- the language (modeling language, specification language, diagrams, . . . ) used to specify the view point

7.3.7.3 Optional elements of a view point specification

Optionally a view point specification may also contain

- **consistency and/or completeness checks** which can be used to validate views for that view point,
- **Evaluation or analysis techniques** which can be used to generate a view complying to that view point.
- **Heuristics, patterns or guidelines** which aid in the synthesis of an associated view.

7.3.7.4 View point libraries

View points need not be specific for a particular type of system or for an architecture analysis and design method. One can reuse standard view points across frameworks and methods. Such generally useful view points are commonly stored in and sourced from view point libraries.
7.3.8 IEC/ISO/IEEE 42010 metamodel

The IEC/ISO/IEEE 42010 metamodel introduces concepts and semantics within which the requirements for a software architecture description are specified.

7.3.8.1 The conceptual realm of the metamodel

Figure 7.7: The core realm of the IEC/ISO/IEEE 42010 metamodel.
7.3.8.2 The core realm of the metamodel

![Figure 7.8: The core realm of the IEC/ISO/IEEE 42010 metamodel.]

7.3.8.3 Architecture description elements and correspondence

An *architecture description element* is an element of an architecture description. Examples include a stakeholder, a concern, a view, and a tactic.

![Figure 7.9: The core realm of the IEC/ISO/IEEE 42010 metamodel.]

A *correspondence* is a relation between elements for an architectural description and a *correspondence rule* is a rule which enforces certain relationships between architecture description elements.
7.3.8.4 The rationale realm of the metamodel

![Diagram 7.10: The core realm of the IEC/ISO/IEEE 42010 metamodel.]

7.3.8.5 The architecture framework of the metamodel

The architecture framework of the metamodel specifies common practices for creating, interpreting, analyzing and using architecture descriptions.

![Diagram 7.11: The core realm of the IEC/ISO/IEEE 42010 metamodel.]

Architecture frameworks are developed for a particular domain of applications or for a specific stakeholder community. Examples of architecture frameworks include MODAF, TOGAF, Kruchten’s 4+1 View Model, RM-ODP.
7.3.8.6 The ADL realm of the metamodel

An **Architecture Description Language (ADL)** is any form of expression for use in architecture descriptions. It facilitates description of architecture model and/or view points according to some architecture description framework.

![Diagram of the IEC/ISO/IEEE 42010 metamodel](image)

Figure 7.12: The core realm of the IEC/ISO/IEEE 42010 metamodel.

Examples of ADLs: include AO-ADL, ACME, xADL, Rapide, SysML, and many others.

### 7.4 Architecture template documents

#### 7.4.1

##### 7.4.1.1 Title page

The title page should contain

- **Title**: Architecture Requirements Document: Project Name
- **Version**: Version identifier
- **Data**: Release date of current version of document
- **Client**: Client name
- **Author**: Name of architect, architecture team or organization doing the software architecture.

##### 7.4.1.2 Document information

- **Revision history:**
- **Document description:**
  - Discuss the purpose of the document itself.
  - Highlight that the software architecture requirements are the non-functional requirements including
    * the architectural scope comprising the responsibilities,
    * the quality requirements for the software system,
    * the access and integration requirements, and
    * the architectural constrains.
7.4.1.3 System context

In this section provide the context of the software system including

- the core purpose of the system,
- who requires the system,
- the stakeholders of the system and their core interest in the system
- the functional scope of the system
  - Including a high-level use case diagram.

7.4.1.4 Architecture scope

In this section you specify the scope of the architectural responsibilities

- i.e. the responsibilities which need to be addressed by the software architecture.
- Examples of architectural responsibilities include
  - providing a concurrent processing environment,
  - providing a temporary or permanent storage environment,
  - monitoring devices,
  - scheduling,
  - providing an integration environment,
  - providing a physics engine,
  - scheduling,
  - file streaming,
  - ....

7.4.1.5 Quality requirements

This section specifies the quality requirements around the functionality offered by the system.

- Quality requirements include
  - e.g. scalability, performance, reliability, security, auditability, integrability, flexibility and extendability, usability, monitorability, cost, ...
- For each quality requirement specify
  - the stakeholder who requires it
    * e.g. the user, the client, the system maintainers, the auditors, ...
  - the context in which that quality is required,
  - the detailed requirement
    * i.e. a quantification or other measurable specification of the requirement.
7.4.1.6 Access and integration requirements

In this section specify

- the different links/channels supported by the system including
  - the access channels through which users can use the system,
    * e.g. the web, mobile device, web services, remote control, hand gestures, ...
  - the channels through which the system integrates with other systems,
  - the channels through which the architectural components integrate,

- For each channel include the specification of
  - any communication protocols which must be used,
  - the quality requirements required for communication across that channel
    * e.g. security, scalability, reliability, performance, auditability, …, requirements

7.4.1.7 Architecture constraints

This section specifies the constraints placed by the client on the architecture

- These are non-negotiables from the client side.
  - Try and discourage client to specify their ideas for the software architecture as architectural constraints.
  - Let them provide their ideas, but these must be taken through the architectural design and validation process.

The architectural constraints may include constraints on

- technologies and frameworks which must be used,
- access/integration channels which must be made available/used,
- hardware infrastructure (e.g. servers, networking infrastructure, …)

7.4.2 Software architecture documentation template

This is a guideline for developing a software architecture description (design document). It is meant to provide some guidelines as an alternative to the Kruchten 4 + 1 approach to documenting a software architecture and is more aligned with the approach of a software architecture providing the infrastructure within which application functionality is deployed and executed.

Throughout the document you are encouraged to use diagrams to illustrate aspects of your software architecture. You can leave out sections which are not relevant to you, add sections which are and generally modify the structure of the document to suit your specific project.

Note, that in addition to the guidelines provided in this document, your document also adhere to the the ISO/IEC/IEEE guidelines of an architectural description.
7.4.2.1 Documentation across levels of granularity

The software architecture should be specified across levels of granularity, i.e. You start with the software architecture for the system as a whole as initial component and recursively decompose it into lower level architectural components.

For example, within AUTOSAR you would specify the architecture of AUTOSAR as a whole, specifying the high-level responsibilities (host application components, provide integration infrastructure for application components, provide bridge to microcontroller, ...) assign them to architectural components (application container, virtual function bus, microcontroller adapter, ...), specify the connectors and structural constraints (potentially in the form of structural patterns like layering in AUTOSAR), specify the architectural tactics/strategies for the high-level component and, if the component directly hosts application components, the concepts and constraints within which the application components are to be developed.

For each level of granularity you should consider providing the following views:

- A structural view specifying the sub-component and internal structure of the component,
- A tactics view specifying the architectural tactics applied to the component,
- If the component hosts application components, an application component concepts and constraints view specifying the concepts and constraints the architecture introduces for the software components which are to be deployed within the software architecture,
- if the component is to be based on some reference architecture, a reference architecture view specifying the reference architecture used, which features of the reference architecture are used and how the architectural components, patterns and tactics map onto those specified in the reference architecture.
- Optionally a technology view specifying the technologies and frameworks.

7.4.2.2 The structural view

The structural view for an architectural component at a particular granularity should specify

- the architectural style or pattern selected to constrain the infrastructure of the component
  - (e.g. layering, microkernel, blackboard, ...)
  - and the rationale for using that pattern,
- the architectural components and the responsibility assigned to each,
- the mapping of components onto the architectural pattern components
  - e.g. the component might map onto the mikrokernel of the architectural pattern
- the connectors between the components.
7.4.2.3 The tactics view

In this view you specify the architectural tactics through which the quality requirements for the architectural components are to be concretely addressed.

For each chosen tactic (including integration patterns), specify

- why the tactic was chosen (e.g. what quality requirement is is meant to address),
- how the tactic affects other quality requirements,
- how the tactic is applied to the component.

Note: Tactics view can be documented effectively using AO-ADL.

7.4.2.4 The application concepts and constraints view

If your component hosts application concepts (e.g. a EJB container hosts enterprise beans, a space in the space-based hosts domain objects and a services container in SOA hosts services), specify the concepts and constraints introduced by the software architecture.

Examples of concepts are

- procedures,
- classes,
- Components as in EJB and other component-based frameworks,
- functions as in functional programming,
- services as in SOA

Examples of constraints introduced by the software architecture include

- that application components must be self-healing,
- that a service must be stateless (as in SOA),
- that a function may not have any side-effects (as in pure functional programming languages like Haskell),
- that a service or component must implement an interface (e.g. CORBA objects implementing IDLs, Web services implementing WSDLs, enterprise beans implementing local and remote interfaces, . . . )
- that application components can interact only via some integration bus (as in AUTOSAR).

7.4.2.5 The reference architecture view

If the component architecture is based on a reference architecture, specify

- which reference architecture is used,
- the mapping of software architecture components are onto the reference architecture components, and
- the mapping of selected tactics onto features of the reference architecture.
7.4.2.6 Technologies and frameworks view

In this view you specify

- technologies including
  - protocols,
  - broker implementation (e.g. a particular CORBA or web services broker),
  - message queue implementation,
  - XML parsers,
  - ...

- Any frameworks you are using (e.g. a particular Java-EE application server like JBoss or Apache Geronimo)
Chapter 8

Software architecture recovery

8.1 What is software architecture recovery?

Definition 8.1.1. Software Architecture Recovery is the process of recovering a software architecture description from an existing system for which there is no authoritative architectural description.

Note that the scope, method, inputs and outputs are all influenced by choice of software architecture definition. In either case, the output should be an IEC/ISO/IEEE 42010 compliant architectural description.

8.2 Why perform a software architecture recovery?

Architecture recovery projects are initiated when there is no valid software architecture description available. The reasons for no longer having an authoritative architectural description may include

- lack of architectural definition and control often in the context of cowboy-style development in the disguise abuse of abused agile processes,
- architectural drift caused by undocumented architectural evolution in order to address architectural concerns,
- architecture erosion in the context of lack of documentation, understanding, and/or buy-in into the designed architecture leading to non-adherence to software architecture specification.

The recovered architectural descriptions are commonly required for

- software architecture assessment for consistency and against non-functional requirements,
- preventing further architecture erosion,
• software architecture improvement, and
• software architecture reuse.

8.3 Challenges of software architecture recovery

Challenges software architects typically face when performing software architecture recovery include

• The system may have a lot of code and a huge code bulk may have to be manually analyzed.
• There may be a high level of system heterogeneity, i.e. different aspects of the system designed and built against very different software architecture and technologies.
• The architecture is not explicit in the source code and hence we cannot automate many aspects of the architecture recovery and in particular the extractions of abstractions like patterns and tactics.
• Third party components used in the software system may address architectural concerns and may significantly affect the quality attributes of the system. It may be that neither the source nor sufficient architectural descriptions is available for these components.
• No standard description framework or language for documenting a software architecture.
• Political issues within organizations

8.4 How would you recover a software architecture?

Architecture recovery is complex and following just single approach often leads to dead-ends, incomplete coverage, and omission of core architectural features. To combat these one often combines approaches. Commonly one starts with interviews and available documentation. Then one may perform some request tracing, tracing requests from different channels including batch channels as well as requests for services from different responsibility domains. One may then explore some of the architectural responsibilities which are not sufficiently covered by request tracing. This might cover concerns around security, logging, persistence, adapters, or thread management.

Ultimately one will have to perform abstractions into architectural components with architectural responsibilities (e.g. routing, Demarshalling, Caching, Execution container, Adapter, ...), architectural patterns, architectural tactics (incl integration patterns), and concepts and constraints the architecture introduces for application components.

One will additionally have to analyze artifacts available for third-party components. Typically the above steps are not performed strictly sequential, but one will regularly revisit certain aspects.

As you go along, the understanding of the recovered architecture will have to be fed into an architectural description.
8.5 Steps of architecture recovery

Architecture recovery processes have commonly the following high-level steps:

1. Extraction where the architectural information is extracted from artifacts like the code, vendor component descriptions, configuration files, . . . .

2. Abstraction where architectural significant elements are abstracted into responsibility domains and component abstractions, patterns, tactics, and concepts and constraints of application components.

3. Description during which the software architecture description or specification is generated.

8.6 Approaches to software architecture recovery

Widely used approaches to software architecture recovery include component, interface and connector recovery and request tracing.

Recovery of components, interfaces and connectors can be largely automated and the outputs can be captured by most ADLs. The approach does require the ability to group components into higher-level components. Some work has been done around automating this (see [?] and [?]), but these tools are not yet in practical use.

Request tracing is particularly useful for software architecture recovery and there is good tool support for them (see for example, InTrace [?] and BTrace [?]). Once one has request traces through systems, one can use these traces to identify architectural components along the trace. These identifications and the abstractions into patterns and tactics require, however, manual processing.

8.7 A Systematic Method for software Architecture Recovery (SyMAR)

The Systematic Method for software Architecture Recovery (SyMAR) is a manual method with some options for automation guiding software architects to recover a software architecture description across levels of granularity. The method yields as abstractions

- architectural components addressing architectural responsibilities,
- infrastructural and integration patterns,
- tactics used to address quality requirements, and
- concepts and constraints for application components.
8.7.1 Overview of SyMAR

Figure 8.1: The systematic method for software architecture recovery yielding, for each architecturally significant component, one or more Request Trace Views (RTVs), a Responsibility Allocation View (RAV), a Structural View (SV), a tactics view, Framework Mapping View (FMV) and an Application Concepts and Constraints View (ACCV).

8.7.2 Request traces

To generate request traces one should use representative services/use cases with different integration requirements which are accessed via different access channels including synchronous and asynchronous human and system access channels. Examples include web, mobile, email, web service, message queue, . . . , batch, . . .

The generation of request traces can be automated using tracing tools, but the result needs to be pruned to reflect the request trace at the appropriate level of granularity. However, the full trace information is retained to provide tracing information for lower levels of granularity.
Figure 8.2: An example of how a request trace can expose architecturally significant components – only the last component contains application logic.
Chapter 9

Software architecture analysis

9.1 Overview

Architecture analysis involves analyzing either an existing software architecture or a proposed software architecture specification for its ability to address the non-functional requirements for some system.

An architecture analysis is commonly used to identify architectural weaknesses of either an existing system or of a proposed software architecture and to compare different software architecture proposals.

9.1.1 When is a software architecture analysis done?

An architecture analysis is commonly done either

- after software architecture proposal(s) have been received and before deciding on a particular architecture and commencing with an iterative implementation of system,
- after a software architecture recovery has been done to analyze the recovered software architecture for its strengths and weaknesses,
- before re-architecting a software system in order to modify the software architecture to provide different quality attributes without changing system functionality itself,
- when the software architecture of a system is no longer sufficiently understood and in order to grow common understanding and appreciation of the software architecture in order to achieve a higher level of architecture compliance.

9.1.2 Basic approaches to software architecture analysis

There are two main approaches which are commonly used to perform a software architecture analysis or evaluation:

1. Inspection/analytic techniques involving the critical analysis of architectural decisions.
2. Measurement techniques which use quantitative measurement to assess to what extend a software

9.1.3 Architecture analysis methods

We have a wide selection of architecture analysis methods to choose from. Here is a high-level summary with the focus of each of the methods:

1. **SAAM** (Software Architecture Analysis Method) is used to analyze how functionality is allocated across structure, and also includes an analysis of how patterns and tactics are used to address quality requirements.

2. **ATAM** (Architecture Trade-off Analysis Method) is a further development of **SAAM**. The method adds more structure to explicitly expose how quality requirements traded off. This is one of the most widely used methods in industry.

3. **SAAMCS** (SAAM for Complex Scenarios) is a method for identifying complex scenarios and to analyze how these impact the architecture.

4. **SACAM** (Software Architecture Comparison Analysis Method) extends **ATAM** by quantifying the assessment through a weighting system which enables one to compare architectural candidates.

5. **SBAR** (Scenario-Based Architecture Re-engineering) is both, an analysis and a re-engineering methodology. It is used to identify areas of quality concerns and defines way of evolving an architecture to address these.

6. **ESAAMI** (Extending SAAM by Integration in the Domain) which operates more at a functional level, focusing on identification components for re-use and introducing scenario abstractions (proto-scenarios) encapsulating scenario commonalities. The amount of reuse is fed back into scenario weighting for architecture assessment.

7. **CBAM** (Cost-Benefit Analysis Method) which includes cost quality requirement (and in trade-off).

8. **ALPSM** (Architecture Level Prediction of Software Maintenance)

9. **PASA** (Performance Assessment)

10. **ALMA** (Architecture-level modifiability analysis)

11. **ALRRA** (A methodology for architecture-level reliability risk analysis)

9.2 The Architecture Trade-off Analysis Method (ATAM)

The Architecture Trade-off Analysis Method (ATAM) is a systematic method for identifying architectural decisions, the rationale behind them, and the trade-off decisions made.

The method has been developed by Bass, Clements, Katzman and Klein from the Software Engineering Institute (SEI) at the Carnegie Mellon University [?].
9.2. THE ARCHITECTURE TRADE-OFF ANALYSIS METHOD (ATAM)

9.2.1 Benefits of ATAM

Benefits of ATAM include that the method

- exposes precise quality requirements,
- provides systematic approach for assessing architecture against architecture requirements, and
- assists with growing a common understanding of the architecture across stakeholders.

9.2.2 Applicability of ATAM

ATAM can be used to assess

- a proposed architecture,
- an existing architecture, and
- proposed architectural modifications.

9.2.3 Overview of the ATAM process

![Figure 9.1: An overview of the ATAM method.](image)

9.2.4 The role players in ATAM

There are three core groups who participate in an ATAM based architecture assessment project:
1. The evaluation team.
2. The project decision makers.
3. The architecture stakeholders.

9.2.4.1 The evaluation team

The architecture evaluation team should not include members who are part of the project team, i.e. who have a stake in the architecture. The team often includes external consultants. This can assist with avoiding political issues and with obtaining an independent, external assessment. The evaluation team should preferably have at least two members.

Skills requirements for the evaluation team should include that of having a solid background in architecture concepts including architecture requirements, patterns, strategies and reference architectures, and understanding the technologies and frameworks used within the proposed/existing architecture. Furthermore, the evaluation team should have good analytical, facilitation and presentation skills.

9.2.4.2 Project decision makers

The project decision makers are the people who make decisions about the project including strategic, investment and logistic decisions.

They typically include the person(s) or team which proposed the project (e.g. the product champion), the project sponsor, and project management.

9.2.4.3 Other architecture stakeholders

Additionally the role players in an ATAM based architecture analysis may include any further parties who have an interest in the architecture and the resultant system. This may include

- the project’s architecture team,
- developers,
- integrators,
- testers,
- maintainers,
- the operational team,
- users, and
- external parties the system is supposed to integrate with.

9.2.5 Outputs of ATAM

The outputs of ATAM include the following:
9.2. THE ARCHITECTURE TRADE-OFF ANALYSIS METHOD (ATAM)

1. A concise presentation of the architecture focusing on core components, their responsibilities and interactions, patterns, strategies and reference architectures employed, frameworks and technologies used, as well as the integration infrastructure.

2. A re-affirmation of the Business vision and goals for the project — often this is simply a copy of the vision & scope document.

3. The quality and integration requirements in the context of business goals.

4. A mapping of architectural decisions onto quality requirements exposing rationale of architectural decisions.

5. A set of identified sensitivity and trade-off points exposing architectural decisions which make trade-off across quality requirements.

6. A set of risks and non-risks including architectural decisions which might have business impact and ones which were thought to be risks, but have been shown to be safe.

9.2.6 The ATAM process

The ATAM process has the following four phases:

1. An initiation phase during which the team is prepared to participate in the ATAM bases architecture analysis project.

2. An elicitation phase during which the requirements and architectural decisions are elicited.

3. An elaboration phase during which the architectural decisions are analyzed.

4. An evaluation phase during which the conclusions about the architecture assessment are drawn. It is in this phase that the evaluation report is finalized.

9.2.6.1 The initiation phase

An ATAM process is initiated with a request for an architectural analysis. The first step is to identify and commission architecture evaluator or evaluation team and brief them on the reason for performing the architecture evaluation. Subsequently the role players participating in the process are identified and the team is assembled.

Once the team has been assembled the ATAM concepts are presented and explained to the team. This includes the aim or purpose of the architecture evaluation, an explanation of the ATAM process and the rationale behind the process, the responsibilities of the different role players, and an explanation of the ATAM phases. For each phase one discusses the purpose, the inputs and outputs, and the activities performed.

Team members are encouraged to raise questions or concerns around the process.

9.2.6.2 The elicitation phase

The elicitation phase has the following sub-steps:

1. The architecture evaluation team presents the ATAM in more detail.
CHAPTER 9. SOFTWARE ARCHITECTURE ANALYSIS

2. Business presents the business drivers.
3. The project’s architecture team presents the architecture.
4. Architectural approaches are identified and discussed.
5. A quality attribute utility tree is generated.
6. The architectural approaches are discussed.

9.2.6.2.1 Present the business drivers  The specification of the business drivers should include

- The vision the organization is trying to realize with the project.
- The business objectives.
- Identification of the various stakeholders in the system.
- The financial context including the return on investment estimation.
- The scope of the system including the high level use cases and responsibilities, and the core limitations and exclusions.
- The architectural drivers (the core quality requirements).
- The core integration requirements.
- Any constraints placed on the architecture.

9.2.6.2.2 Present the architecture  The lead architect provides the essence of the architecture in a concise, structured way. The focus should be on core architectural decisions and not on auxiliary details.

Aspects which should be covered in the architecture presentation include

- the context/environment of system,
- any technical constraints including constraints around the deployment environment, middleware technologies, skills availability, and so on.
- the architectural approaches followed including the architectural patterns chosen, the architectural strategies used, any reference architectures used, and the framework and technology decisions made.

If available, an architectural description is provided to the architecture evaluation team. This would be particularly useful if it was in a IEC/ISO/IEEE 42010 compliant form.

9.2.6.2.3 Identify architectural approaches  During this phase there are generally extensive discussions between the project’s architecture team and the architecture evaluation team in order to deepen understanding of the architectural decisions made, and the rationale behind those architectural decisions.

Usually further architectural decisions are identified during this phase. Some, but not necessarily all of these might not have been made explicitly.
9.2.6.2.4 Generate quality attribute utility tree  The quality attribute utility tree is a tree which has

- quality attributes as 1’st level,
- refined attributes with utility and quantification as 2’nd level.
- an concrete scenarios which require that quality attribute as 3’rd level

<table>
<thead>
<tr>
<th>Quality attribute</th>
<th>Attribute refinement</th>
<th>Scenarios</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modifiability</td>
<td>Flexibility to modify confidence level matching.</td>
<td>Modify field weightings.</td>
</tr>
<tr>
<td></td>
<td>Flexibility to change data cleansing algorithm.</td>
<td>Replace rules engine for short list selection.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Modify required confidence levels</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Modify confidence level matching algorithm.</td>
</tr>
</tbody>
</table>

Table 9.1: An examples quality attribute utility tree.

The tree is commonly rendered as a table as shown in table

During this step scenarios from attribute utility tree are prioritized and analyzed in order of their priorities. The project’s architecture team explains how the architecture supports each scenario.

The evaluation team identifies and documents architectural decisions with their risks and non-risks, sensitivity points, and trade-offs.

9.2.6.3 The elaboration phase

During the elicitation phase one captures information about the software architecture and develops an understanding without assessing the software architecture.

During the elaboration phase one assesses the architectural decisions, identifying risks not identified by project team, and weighing up against alternatives. Nevertheless, one still obtain further information from stakeholders to deepen understanding.

Typically one performs a an analysis of the code of critical components to obtain an independent understanding of key features of the software architecture.

One then brainstorm, prioritizes and analyzes further scenarios, expanding the attribute utility tree.

Ultimately one needs to assess the trade-offs made in the context of the business requirements. Note that one should not forget investigating the maintainability of the system.
9.2.6.4 The evaluation phase

During the evaluation phase the architecture evaluation team writes the evaluation report. The report should include

- a management overview of the findings,
- detailed explanations of the rationale behind the findings,
- recommendations for changes and the rationale for them,
- cost estimation and discussion for the benefits of the proposed changes,
- a migration plan for the changes to be applied, and
- a list of open risks which should be verified through testing.
Chapter 10

The Model Driven Architecture (MDA)

10.1 Overview of Model-Driven Architecture

The Model-Driven Architecture (MDA) is a framework of standards for model-driven engineering maintained by the Object Management Group (OMG). The standards are implemented by various MDE tools. Effectively MDA provides a practical architecture for Model-Driven Development (MDD), i.e. an infrastructure addressing the quality requirements of MDD.

10.1.1 Architecture and Application Design within MDA

Figure [10.1] depicts the decoupling in MDA of the technology and architecture neutral application design done by domain specialists (e.g. business analysts) from the design of the technical software architecture done by software architects. In the case of enterprise systems development, the platform independent model (PIM) is the business model containing the domain entity, services contracts and business process specifications for the business. This business model is preferably generated and maintained by business analysts across the organization, contributing business knowledge from different business domains to the organization-wide business model.

10.1.2 Problems MDA aims to address

Domain models contain technology and architecture neutral requirements and design specifications. They can be developed, owned and maintained by domain specialists like business analysts.

The value extracted from such an approach includes simpler management of requirements and technology-neutral designs, the ability to perform model validation to ensure model quality, and the ability to generate a range of artifacts including the system code, documentation and system tests resulting in cost reduction in improved consistency.
Furthermore, since the models are technology and architecture neutral, they have better longevity, surviving technology and architecture changes. In the case of technology and architecture changes new system, documentation and test artifacts can be generated from the technology and architecture neutral models. This results in increased flexibility and the ability to quicker (and at lower cost) extract value from technology advances.

10.1.3 MDA standards

- **Metamodel** specification
  - (Essential) Meta-Object Facility: MOF / EMOF

- **Modeling languages**
  - Generic → UML
  - Domain-specific → DSL

- **XML standard** for metadata & model encoding
  - *XML Metadata Interchange* (XMI)
10.1. OVERVIEW OF MODEL-DRIVEN ARCHITECTURE

- **Constraint specification:**
  - *Object Constraint Language* (OCL)

- **Model-To-Model Transformation**
  - *Query/View/Transformations*
    - *Declarative* → *QVT-Relational*
    - *Operational/algorithmic* → *QVT-Operational*

- **Text/code generation**
  - *MOF Model to Text* (MOFM2T)
  - Better (?) approach: Define text syntax for DSL
    - *Render model in text syntax*

10.1.4 Model-to-model transformations

In model-to-model transformations one has metamodels for the source and target domains as well as the source model itself. One transforms a source complying to the metamodel for the source domain to a target model complying to the metamodel for the target domain.

10.1.5 MDA tools

MDA is supported by a set of tools which either implement MDA standards, or are tools which do not implement standards.

10.1.5.1 MDA standards implemented by MDA tools

- A language to specify languages ⇒ MOF or EMOF.
  - Meta-Object Facility & Essential MOF.
- A generic modeling language ⇒ UML
  - the Unified Modeling Language.
- The possibility to use domain-specific/specialized languages ⇒ DSL.
- The ability to specify constraints at metamodel and model levels and the ability to query models ⇒ OCL.
  - *Object Constraint Language*
- The ability to specify model-to-model transformations
  - through declarative transformation declarations ⇒ QVT-R
    * Query-View-Transformations Relational
  - through imperative algorithms ⇒ QVT-O
    * Query-View-Transformations Operational
- The ability to transform models into text (e.g. code) ⇒ MOFM2T or M2T.
  - MOF Model-to-Text or Model-to-Text.
10.1.5.2 MDA tools not based on MDA standards

In MDA MOF/EMOF based metamodels are used to specify the abstract language within which a model (e.g. a domain model) is specified. However, MDA does not yet specify any standards for specifying a concrete syntax for the abstract language. The concrete syntax could be a text or a graphical/diagrammatic syntax.

There are however a range of tools provided in modeling frameworks (particularly the Eclipse Modeling Framework, EMF) for specifying a concrete text or diagrammatic syntax for a language and for generating a language aware text or diagrammatic editor. For specifying a concrete text syntax and generating a syntax aware text editor one can use either XText or EMFText. For specifying a concrete diagrammatic syntax and generating a diagrammatic editor one can use the Graphical Editing Framework (GEF) and the Graphical Modeling Framework (GMF).

10.2 MDA tools

The Model Driven Architecture (MDA) specifies an extensive set of standards for tools to specify

- the semantics of a language,
- constraints across object graphs,
- to store models in XML, and
- model transformations either declaratively or imperatively.

Additionally we have tools to specify either

- a concrete text syntax, or
- a concrete diagrammatic syntax.

10.2.1 Eclipse modeling tools

The Eclipse Modeling Project provides an implementation of most of the MDA standards. Examples of the tools implemented within Eclipse Modeling include

- Ecore → implementation of EMOF
- Model Query and Model Validation
  - make use of Eclipse OCL Engine
- XText & EMFText
  - for specifying concrete text syntax,
  - generate validating editor from text syntax definition.
- Eclipse Graphical Modeling Framework (GMF)
  - Graphical tooling definition
  - Bi-directional mapping
- Transformations
  - QVTr → implementation of QVT-Relational
  - QVTo → implementation of QVT-Operational
  - Acceleo → implementation of MOFM2T
10.2.2 Overview of ecore

Ecore is Eclipse’s implementation of OMG’s EMOF. It is used to specify the semantics of languages (not a specific concrete syntax). To this end one the concepts and relationships between concepts, other languages. Ecore has a diagrammatic syntax which is similar to UML class diagrams without support for operations. Ecore supports packages, basic data types including enums, classes, attributes, specialization, aggregation and association relationships.

OCL constraints can be applied to Ecore models. Doing this specifies constraints at metamodel level — e.g. for every precondition there must be an associated exception versus for this precondition, that exception will be raised.

10.2.2.1 Ecore example

The URDAD-DSL supports reuse of state constraints across functional requirement, i.e. the same constraint can be both a precondition for one or more services, as well as a post-condition for other services.

10.2.3 Overview of XMI

XML MetaData Interchange (XMI) can be used for any metadata whose metamodel can be expressed in MOF or EMOF. It is commonly used to store models in tool and vendor independent way. This could be, for example, UML models or models encoded in some domain specific language (DSL). Note also that the metamodels themselves are encoded in XMI.

10.2.3.1 XMI example

As an example, consider the following snippet of the URDAD-DASL metamodel encoded in XMI:

```xml
<subimport/>
```

Listing 10.1: Part of URDAD metamodel in XMI.
10.2.4 Overview of the OCL

The Object Constraint Language (OCL) is a constraint language used to specify constraints which are applied across an object graph. To this end one may navigate association relationships (including aggregation and specialization relationships).

OCL can be applied to any modeling language which is based on a MOF/EMOF metamodel, it can thus be used for UML models as well as for any DSL.

OCL supports the specification of invariance constraints on objects and the specification of preconditions and postconditions on services/methods. It can be applied either at instance model level to specify a constraint in the modeling domain (e.g. savings accounts must have a positive balance) or at meta-model level to specify constraints for instance models (e.g. in any instance model every pre-condition must have associated an exception which will be raised if that precondition is not met).

10.2.4.1 Examples

10.2.4.1.1 Invariance constraints

Invariance constraints are state constraints applied across an object graph. The starting point for the navigation across the object graph is a particular class.

For example, one can use the following constraint to specify that a savings accounts may not have a negative balance

```
context SavingsAccount
inv: self.balance >= 0
```

Listing 10.2: Part of URDAD metamodel in XMI.

The following, more complex constraint demonstrated the navigation across the object graph following association (incl specialization) links. The constraint specifies that the sum of the amounts of all transactions made on an account must be equal to the balance of an account:

```
context Account
inv: self.transactions->collect(amount)->sum() = self.balance
```

Listing 10.3: Part of URDAD metamodel in XMI.

10.2.4.1.2 Precondition examples

Preconditions represent the conditions under which a service provider is not obliged to provide a service. For example, one could use the following constraint to specify a precondition for the debit service is that requires that the resultant balance may not be less than the minimum balance

```
context Account::debit(amount:Real): TransactionConfirmation
pre sufficientFunds: amount <= self.balance - self.minimumBalance
```

Listing 10.4: Part of URDAD metamodel in XMI.

If the pre-condition is not met, the service may be refused. This this refusal is typically communicated by raising an exception which is associated with that precondition and since the contract made provision for refusing the service under these
conditions, that refusal (and the catching of the associated exception) do not represent errors in the service itself.

### 10.2.4.1.3 Postcondition examples

Post-conditions are constraints which need to be met after a service has been provided. They are assigned to object methods.

For example, to specify one can use the following constraint to specify that debiting an account with an amount must reduce the balance with just that amount, i.e. that the transaction fee must be raised in a separate transaction:

```
1  context Account::debit(amount: Real) : TransactionConfirmation
2  post: balance = balance@pre − amount
3  post: self.transactions->size() = self.transactions@pre->size() + 1
```

Listing 10.5: Part of URDAD metamodel in XMI.

Post-conditions may also be applied to the return value of a service. For example, to specify that after having found the numerical root of a function, the function value at that numerically calculated root must be close to zero, we can use:

```
1  context RootSolver.findRoot([f:Function, initialGuess:Real, eps:Real]): Real
2  post: f.value(result).abs() < eps
```

Listing 10.6: Part of URDAD metamodel in XMI.

OCL has a rich semantics and also allows one to specify that during the realization of a service, a particular service must be called. This is useful to specify constraints at a particular level of granularity and can be seen analogous to using mock objects when unit testing a service.

For example, we can use the following constraint to specify that during processing a claim the determineClaimCoverage message must have been sent:

```
1  context Claims::processClaim(claim : Claim) : ClaimSettlementReport
2  post: self.policies^determineClaimCoverage(claim : Claim)
```

Listing 10.7: Part of URDAD metamodel in XMI.

### 10.2.5 Overview of QVT-Operational (qvt-o)

The Query-View-Transformations – Operational (QVT-O) is an imperative (operational/algorithmic) language for specifying transformations supporting standard imperative constructs like conditionals, iteration, variables, and so on. Consequently it does not support bidirectional transformations and cannot be used for consistency checking of models. It is also more difficult to prove its characteristics formally.

However, for certain problems and for people who feel more comfortable using an imperative approach such an imperative approach to specifying transformations can be simpler.
10.2.5.1 qvt-o example

As an example, consider the following QVT-O snippet which collects all primitive attributes:

```plaintext
query Attribute:getAllPrimitiveAttributes() : Sequence (Attribute)
{
  if self.type.isKindOf(PrimitiveDataType)
    then
      Sequence
      { 
        object Attribute 
        { 
          name := self.name;
          type := self.type;
        }
      }
    else 
      self.type.asType(Class).attributes
      ->getAllPrimitiveAttributes()
      ->flatten() -- flatten nested collections into single level collection
    endif;
}
```

Listing 10.8: Part of URDAD metamodel in XMI.

10.2.6 Overview of QVT-Relational (qvt-r)

10.2.6.1 Overview of QVT-Relational

Query-View-Transformation – Relational (QVT-R) is a declarative language for specifying model-to-model transformations (similar to XSLT). Being declarative, it is bi-directional and can be used for consistency checking of two models, i.e. to check whether a target model is consistent with some source model and vice versa.

10.2.6.1.1 qvt-r example

Consider the following example from a UML-to-RDBMS transformation.

```plaintext
transformation umlRdbms (uml : SimpleUML, rdbms : SimpleRDBMS)
{
  relation ClassToTable { //map each persistent class to a table
    enforce domain uml c:Class {
      namespace = p:Package {};
      kind := 'Persistent';
      name = cn;
    };
    enforce domain rdbms t:Table {
      schema = s:Schema {};
      name = cn;
    };
    when { PackageToSchema(p, s); }
    where { AttributeToColumn(c, t); }
  }
}
```

Listing 10.9: Part of URDAD metamodel in XMI.

The transformation enforces (potentially writes) on both sides. The when clause selects when transformation is enforced and the where clause specifies that the At-
tributeToColumn transformation must be done if the umlRdbms transformation is done.

### 10.2.7 Overview of EMFText

EMFText and XText are frameworks for specifying a **concrete text syntax** using a EBNF syntax specification. One can then generate a **syntax-aware editor** for a DSL which validates the specification against the metamodel (including constraints specified for the metamodel), i.e. one uses the editor to specify, via a concrete text syntax, a model complying to the metamodel.

#### 10.2.7.1 EMFText example

For example, to specify an URDAD services contract one specifies both, functional requirements with pre- and post-conditions and the quality requirements.

For each precondition one specifies the exception which will be used to communicate that the service was not provided due to that precondition not being met. Optionally one can specify an inverse service through which lasting effects of a service can be reversed, i.e. implementing a compensating work flow.

For a functional requirements one specifies **requires** links for traceability of to the stakeholder who requires that functionality (the stakeholder can also be another service). One also specified the data structures for the request and result objects.

```java
ServiceContract enrollForPresentation {
  FunctionalRequirements receiving Variable enrollForPresentationRequest ofType EnrollForPresentationRequest {
    PreCondition enrollmentPrerequisitesMet requiredBy (TrainingRegulator Student)
      raises EnrollmentPrerequisitesNotSatisfiedException checks constraint enrollmentPrerequisitesForPresentationMet with
      ValueOf enrollForPresentationRequest
    PostCondition enrollmentProcessPerformed requiredBy (Student Client TrainingRegulator)
      ensures constraint studentEnrolledForPresentation
    {
      create Variable studentEnrolledRequest ofType StudentEnrolledRequest
      set Query OCL: "studentEnrolledRequest.personIdentifier"
      equalTo Query OCL: "enrollForPresentationRequest.personIdentifier"
      ... }
    PostCondition invoiceIssued ...
  }
  Request DataStructure EnrollForPresentationRequest {
    has identification presentationIdentifier identifying Presentation ...
  }
  Result DataStructure EnrollForPresentationResult {
    has component proofOfEnrollment ofType ProofOfEnrollment ...
  }
}
```

Listing 10.10: Example service contract in textual URDAD DSL syntax.