ROZWÓJ POTENCJAŁU I OFERTY DYDAKTYCZNEJ POLITECHNIKI WROCŁAWSKIEJ

Wrocław University of Technology

Internet Engineering

Tomasz Kubik, Zofia Kruczkiewicz

UML AND SERVICE DESCRIPTION LANGUAGES

Information Systems Modelling

Wrocław 2011

Projekt współfinansowany ze środków Unii Europejskiej w ramach
Europejskiego Funduszu Społecznego

Wrocław University of Technology

Internet Engineering

Tomasz Kubik, Zofia Kruczkiewicz

UML AND SERVICE DESCRIPTION LANGUAGES

Information Systems Modelling

Wrocław 2011

Copyright © by Wrocław University of Technology
Wrocław 2011

Reviewer: Dariusz Caban

ISBN 978-83-62098-22-4

Published by PRINTPAP Łódź, www.printpap.pl

Table of contents

1. Introduction 7
Patterns of multitiered systems 9
2. Introduction to Multitiered Information Systems 11
  2.1. Multitiered Information System 11
  2.2. Software Development Model 13
  2.3. Software Development Process Management 16
  2.4. The role of the Unified Modelling Language - UML 18
3. Overview of design patterns for supporting information systems modelling 20
  3.1. Fundamentals of the design patterns identification 21
  3.2. Creational patterns 22
    3.2.1. Abstract Factory 22
    3.2.2. Builder 23
    3.2.3. Factory Method 24
    3.2.4. Prototype 26
    3.2.5. Singleton 27
  3.3. Structural patterns 28
    3.3.1. Adapter 28
    3.3.2. Bridge 29
    3.3.3. Composite 31
    3.3.4. Decorator 33
    3.3.5. Façade 35
    3.3.6. Flyweight 36
    3.3.7. Proxy 38
  3.4. Behavioural patterns 40
    3.4.1. Chain of Responsibility 40
    3.4.2. Command 42
    3.4.3. Interpreter 42
    3.4.4. Iterator 44
    3.4.5. Mediator 45
    3.4.6. Memento 47
    3.4.7. Observer 47
    3.4.8. State 49
    3.4.9. Strategy 50
    3.4.10. Template Method 51
    3.4.11. Visitor 51
4. Design patterns used to build the Business Tier 53
  4.1. Basic issues of the Business Tier design 53
    4.1.1. Using the session components 53
    4.1.2. Using the Entity components 54
  4.2. Bad practices of the Business Tier design 55
  4.3. Analysis of basic design issues 56
    4.3.1. Business Delegate Pattern 56
    4.3.2. Service Locator Pattern 59
    4.3.3. Session Façade Pattern 61
    4.3.4. Application Service Pattern 64
    4.3.5. Business Object Pattern 66
    4.3.6. Composite Entity Pattern 69
    4.3.7. Transfer Object Pattern 72

4.3.8. Value List Handler Pattern 73
5. Design patterns used to build the Presentation Tier 76
5.1. Basic issues of the Presentation Tier design 76
5.2. Bad practices of the Presentation Tier design 78
5.3. Design cases 79
5.3.1. Intercepting Filter 79
5.3.2. Front Controller 81
5.3.3. Context object 83
5.3.4. Application Controller 85
5.3.5. View Helper 87
5.3.6. Composite View 89
5.3.7. Service to Worker 91
5.3.8. Dispatcher View 93
6. Design patterns used to build the Integration Tier 95
6.1. Basic issues of the Integration Tier design 95
6.2. Bad practices of the Integration Tier design 96
6.3. Analysis of basic design issues 97
6.3.1. Data Access Object 97
6.3.2. Domain Store 100
7. Example of the multitiered web application 106
7.1. Two examples of architectures of the multitier application as the *Visual Web Java Server Pages* applications 106
7.2. The *Visual Web Java Server Pages* application based on synchronization of data by an application 107
7.2.1. Structure of project 107
7.2.2. Business Service Sub-tier 109
7.2.3. Application Service of the Business Tier as the remote sub-tier 112
7.2.4. Integration Tier 113
7.2.5. Presentation Tier 114
XML-based service description languages 119
8. RDF (Resource Description Framework) 121
8.1. Model 123
8.2. Vocabulary 125
8.2.1. RDF vocabulary 126
8.2.2. RDFS vocabulary 128
8.3. RDF serialization 131
8.3.1. RDF/XML 132
8.3.2. Terse RDF Triple Language (Turtle) 137
8.3.3. N-Triples 141
8.4. RDF Applications 141
8.4.1. Dublin Core, FOAF 141
8.4.2. RDF API 142
9. OWL (Ontology Web Language) 145
9.1. Ontology and its languages 145
9.2. OWL overview 146
9.2.1. OWL vocabulary 146
9.3. OWL details 150
9.3.1. OWL header 150
9.3.2. Classes 151
9.3.3. Properties 154

9.3.4. Annotations ... 158
9.3.5. Datatypes and facets ... 159
10. WSDL (Web Services Description Language) ... 162
10.1. Structure of a WSDL document ... 164
10.2. Constructs in WSDL 1.1 ... 166
10.2.1. Element <documentation> ... 166
10.2.2. Element <definitions> ... 166
10.2.3. Element <import> ... 168
10.2.4. Element <types> ... 168
10.2.5. Element <message> ... 169
10.2.6. Element <portType> ... 169
10.2.7. Element <binding> ... 169
10.2.8. Element <service> ... 169
10.3. Sample of WSDL 1.1 document ... 170
10.4. Constructs in WSDL 2.0 ... 171
10.4.1. Element <description> ... 171
10.4.2. Element <documentation> ... 171
10.4.3. Elements <include> and <import> ... 171
10.4.4. Element <types> ... 173
10.4.5. Element <interface> ... 173
10.4.6. Element <binding> ... 176
10.4.7. Element <service> ... 177
10.5. Sample of WSDL 2.0 document ... 177
11. SAWSDL (Semantic Annotations for WSDL and XML Schema) ... 179
11.1. Annotation Mechanism ... 180
11.1.1. Model Reference ... 180
11.1.2. Schema Mapping ... 181
11.2. Annotating WSDL Documents ... 181
11.3. Sample of SAWSDL description ... 182
11.4. SAWSDL API ... 182
12. UDDI (Universal Description, Discovery and Integration) ... 184
12.1. Technical Architecture ... 185
12.2. UDDI data structures ... 187
12.2.1. businessEntity ... 188
12.2.2. businessService ... 189
12.2.3. bindingTemplate ... 190
12.2.4. tModel ... 190
12.2.5. publisherAssertion ... 191
12.2.6. operationalInfo ... 192
12.3. UDDI Interfaces ... 192
12.3.1. Inquiry API Set ... 192
12.3.2. Publication API Set ... 193
12.3.3. Security Policy API Set ... 194
12.3.4. Custody and Ownership Transfer API Set ... 194
12.3.5. Subscription API Set ... 195
12.3.6. Value Set API Set ... 195
12.4. Using WSDL Definitions with UDDI ... 195
13. WS-CDL (Web Services Choreography Description Language) ... 197
13.1. Different views on business processes modelling ... 198
13.1.1. Orchestration ... 199

13.1.2. Choreography 200
13.2. WS-CDL document 200
13.3. package 202
13.3.1. informationType 203
13.3.2. token, tokenLocator 203
13.3.3. roleType 204
13.3.4. relationshipType 205
13.3.5. participantType 205
13.3.6. channelType 206
13.3.7. Choreography 208
13.3.8. Variables 209
13.3.9. Activity notation 210
13.3.10. Ordering structures 211
13.3.11. WorkUnit-Notation 211
13.3.12. Interaction activity 212
13.3.13. Perform activity 213
13.3.14. Assign activity 214
13.3.15. SilentAction activity 214
13.3.16. NoAction activity 215
13.3.17. Finalize activity 215
13.3.18. Exception block 215
13.3.19. Predefined functions 215
13.3.20. WS-CDL Example 216
Literature 222

1. Introduction

This document contains reference materials supporting "INFORMATION SYSTEMS MODELLING, UML AND SERVICE DESCRIPTION LANGUAGES" course offered at Wrocław University of Technology. The language of the course is English. The level of English foreseen for attendees is advanced. Students enrol obligatorily for the course on the second term of the first year, during which 30h of lectures as well as 30h of laboratories take place. Workload is 150, and number of ECTS points equals 5.

Outcome: Knowledge of techniques based on design patterns used in object analysis, design and programming. Web Services architecture design and implementation.

Content: Design patterns of Client, Web, Business and Enterprise information system tiers of object oriented software, XML based service description languages, as WSDL (Web Services Description Language) and SAWSDL (Semantic Annotations for WSDL and XML Schema), RDF (Resource Description Framework) and OWL (Ontology Web Language), UDDI (Universal Description, Discovery and Integration), WS-CDL (Web Services Choreography Description Language).

PART I

Patterns of multitiered systems

2. Introduction to Multitiered Information Systems

Modelling of the Information System is based on mapping of data and processes existing in the real world into data structures and processes of some software domain (Figure 2.1).

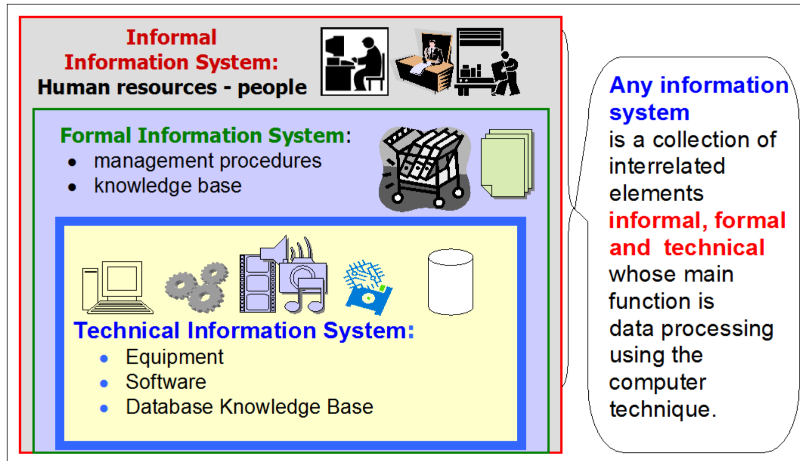


Figure 2.1. Definition of the *Information System*.

An *Informal Information System* is based on human resources. It supports work of people. A *Formal Information System* consists of management procedures and knowledge base as well as clearly outlined functionality of system. The *Technical Information System* is defined as follows:

– An organized team of technical resources (computers, software, hardware teletransmission etc.),

– Used for collecting, processing and transmitting information.

The *Information System* is a collection of interrelated elements informal, formal and technical whose main function is data processing using the computer technique.

The Section 2.1 presents general characteristics of a multitiered information system, the Section 2.2 describes a software development process, the Section 2.3 shows the Software Development Environment and the Section 2.4 characterizes the role of UML (Unified Modelling Language).

2.1. Multitiered Information System

Figure 2.2 shows the multitiered information system needed for a large number of users, huge size of executing data and vast amounts of executing services. Each tier is specialized in the implementation of sub-services used to implement the full functionality of the system. The Busines Tier processes and stores data dedicated to one user, or it uses all data stored in a common buffer of many services called by all users.

In software engineering, scalability is a property of a system, which indicates its ability either to handle growing amounts of work or to become enlarged [1] – it is a feature that allows the behaviour of the system to remain in fact in these situations. For example, it can refer to the capability of a system to increase total throughput under increased load, because it can add resources such as buffers with data and instances of services.

Program performance is defined by a number of units of input data (data size), which in due time, the program manages to transform into units of output data.

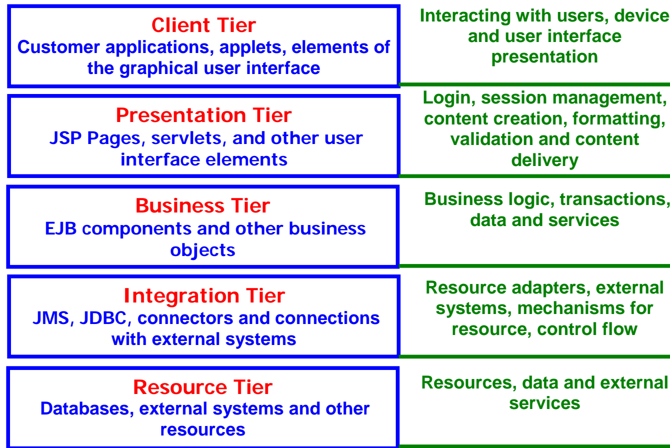


Figure 2.2. Example of the multitier *Information System* [2].

Figures 2.2-2.5 show the multitiered scalable information system with high performance [9].

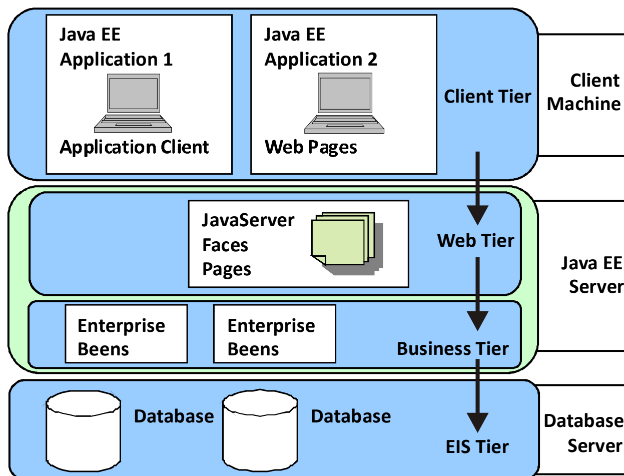


Figure 2.3. Multitiered Applications [9].

Figure 2.3 shows multitiered application: two types of the Client Tier, the Web Tier (Presentation Tier) for the web application, the Business Tier and the tier of Enterprise Information System (EIS Tier).

Figure 2.4 presents the services of the Business Tier independent of the kind of the Client Tier because of a business logic and data encapsulation. The access to the Business Tier is the same for the Presentation Tier and the Application Client.

Figure 2.5 shows access of the Application Client or the Web Pages Client (the Client Tier) across the Web Tier to the EIS Tier.

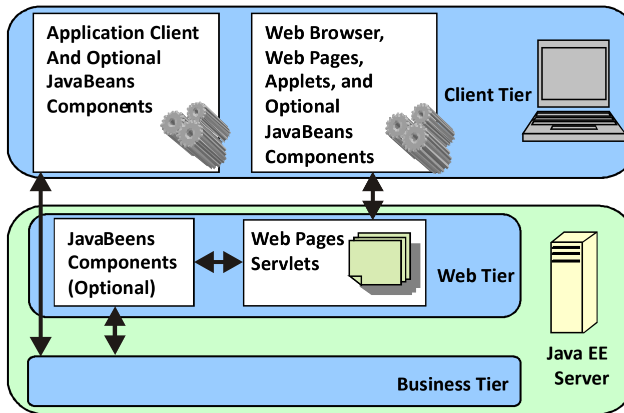


Figure 2.4. Web Tier and Java EE Applications [9].

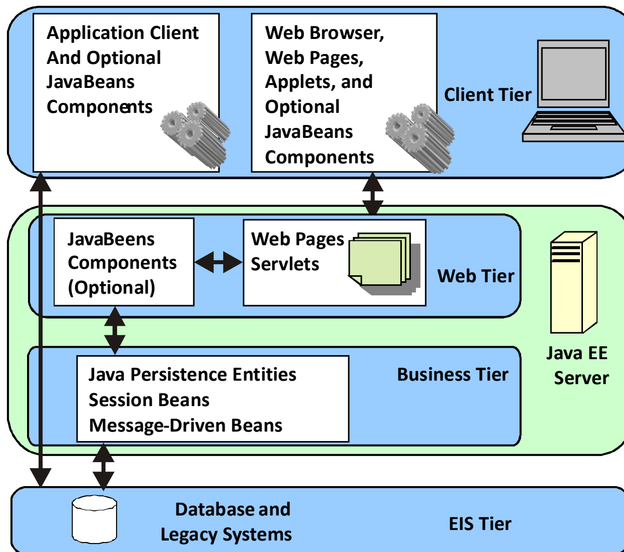


Figure 2.5. Business and EIS Tiers [9].

2.2. Software Development Model

The process model of software development is based on its management model and the software construction (or the software life cycle model) [4], [5], [6], [8].

During the software development, the two questions related to software development should be answered:

– What and how to do? – while constructing the software (Table 2.1),

– When to perform? – during the software development process management (Figure 2.6).

Table 2.1. The software life cycle model [4].

Modelling the structure and dynamics of the system	**Implementation of the structure and dynamics of the system, code generation**	
Perspective of the concept	Perspective of specifications	Perspective of implementation
What to do?	How should I use?	How to perform?
• Model of the real system (business modelling) • Requirements • Analysis (conceptual model) • Conceptual model tests	• Design model (hardware and architecture software; user access; storage) • Deployment model • Design model tests • Deployment tests	• Programming (specification of the program: declarations, definitions; additional data structures: structure of containers, files, databases) • Software tests • Implementation

Table 2.1 presents the software life cycle model as the workflows, which consist of the following activities: business modelling, requirements, analysis, design, implementation, change management, business management, environment and test of the different product of the software life cycle.

Figure 2.6 shows when the workflows execute in the Unified Iterative-Incremental Software Development Process. The workflows develop the models of software and answer the question as follows: when do the tasks of workflows happen?

Workflows are defined as follows: [4]

– Business modelling - a description of dynamics and structure,
– Requirements - requirements specification through use cases,
– Analysis and design - architectural development of different perspectives,
– Programming - software development, unit testing, system integration,
– Testing – to describe test data, procedures and correctness metrics,
– Implementation - to determine the final configuration of the system,
– Configuration management – to gain control over changes and to ensure coherence of the system components,
– Project management – to describe various strategies of an iterative process,
– Determination of the environment – the description of a structure necessary to develop a system.

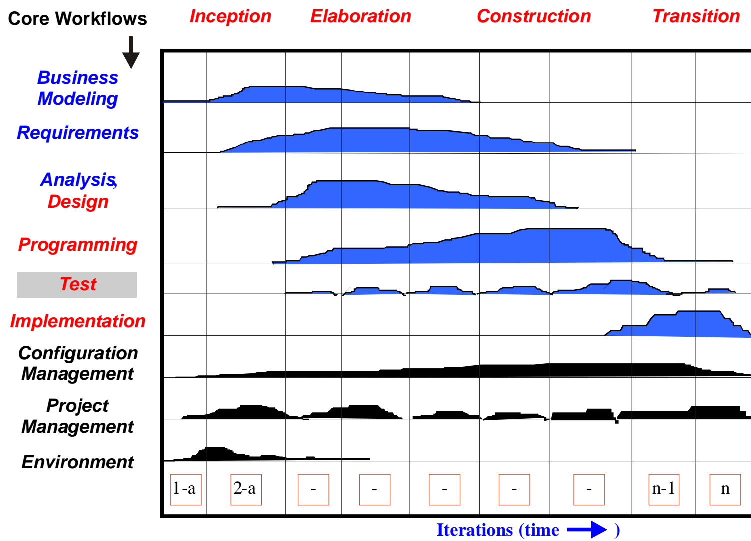


Figure 2.6. Unified iterative-incremental software development process - when? [4].

These are all perspectives on the design of an object-oriented information system [8]:

1. The perspective on concept of the analysis model - it refers to identifying what objects should need to do.
2. The perspective on interface specifications of the design model – it concerns answers how to use objects?
3. The perspective on implementation of the implementation model – it represents answers how to implement an interface.
4. The perspective on creating and managing objects of the implementation model - it means separation of individual subsystems to create objects and facility management.
5. The perspective on use of objects (implementation) - an object A should only use an object B, in other words the object A cannot simultaneously create the object B.

The perspectives useful for understanding objects as object identification are as follows [8]:

1. The perspective on concept of the analysis model – it concerns the facts that the object is a collection of various types of liability.
2. The perspective on specifications of the design model – it represents the object as a collection of methods (behaviours) that may be caused by its methods or other objects
3. The perspective on implementation (source code) - it refers the object code consisting of methods, data as well as interactions.

Perspectives for scaling the system as the creation, management and use of objects [8]:

1. The perspective on creating and managing objects – concerns the changes in the implementation on the objects relating to the factories of objects (creating the objects) and should not affect the management of these objects.
2. The perspective on using objects – represents any change of the implementation of object, which should not require other object implementations to be altered.

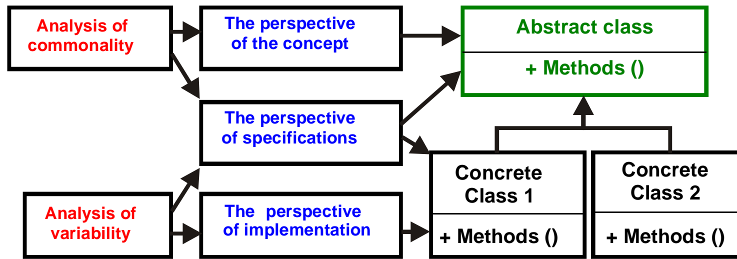


Figure 2.7. Perspectives on understanding objects - the method of identifying objects and classes [8].

Figure 2.7 shows three perspectives of identifying objects and classes as the relationship between the prospect of the specification, design and implementation. During this process, following questions must be answered: what and how to perform [8].

The relationship between the perspectives of specification and concept appoints that the prospect of the specification defines interfaces required to handle all cases of the problem i.e. the common part of data and their behaviours from the viewpoint of the perspective of the concept.

The relationship between the specification perspectives and implementation presents taking into account the specification, we understand how to implement the individual cases (i.e. variable part of data and their behaviours).

The summary of the principles of objectivity are as follows [8]:

– Objects are defined through the prism of their responsibility,
– Encapsulation means any kind of concealment: data, implementation, class (using abstract classes, or interface), the project, the object,
– The use of commonality and variability analysis in order to create abstractions representing the variability in the data and behaviour,
– The use of inheritance as a way of the implementation of the variability in the data and their behaviour,
– Striving for a low degree of relationships,
– Striving for a high degree of consistency,
– Separating code, which uses objects from the code that creates them,
– The principle of a single rule - only one implementation of the operation of a single rule,
– Use of names clearly describing the purpose of objects.

2.3. Software Development Process Management

Figure 2.8 shows the relationship between *People, Project, Product*, and *Process* in Software Development [4].

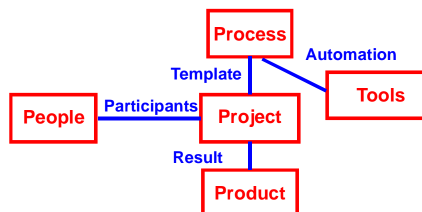


Figure 2.8. The Four Ps: *People, Project, Product*, and *Process* in Software Development [4].

There are important issues of the software development:

- *People*: Architects, developers, testers, users, customers etc.,
- *Project*: Organizational element through which software development is managed ,
- *Product*: Artefacts that are created during the life of the project such as models, source code, executables and documentation ,
- *Process*: A software engineering process is a definition of a complete set of activities needed to transform users' requirements into a product,
- *Tools*: Software used to automate the activities defined in the process.

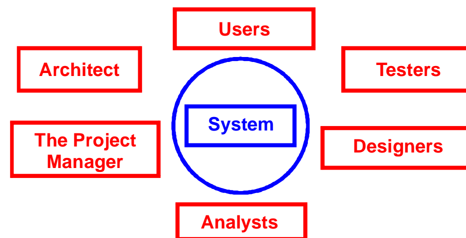


Figure 2.9. Workers participating in the software development [4].

Figure 2.9 shows the *People* as the workers participating in software development. Some workers play the single role as the Architect or the Project Manager and other workers play the multi-types and multi-objects role as users, testers, designers and analysts.

Basic concepts related to *Projects* are as follows:

- Feasibility of the project,
- Risk Management,
- Organization structure of design workers,
- Scheduling project tasks,
- Understanding of the project,
- Rational activities in the project.

The main *Project features* represent:

- Sequence changes in the project,
- Iteration series,
- Organizational Pattern.

The *Products* represent the following things:

- System as the collection of models such as Use-Case Model, Analysis Model, Design Model, Deployment Model, Implementation Model, Test Model (Figure 2.10),
- Diagrams: class, interaction, cooperation, states,
- Requirements, tests, manufacture, installation,
- System composed of artefacts representing programming tools, compilers, computers programmers architects testing facilities traders administrators.

Artefacts are the general terms for any kind of information created, produced, changed, or used by workers in developing system [4]. There are artefacts related to software creation (requirements, analysis, project, programming, tests) and artefacts of the project management process.

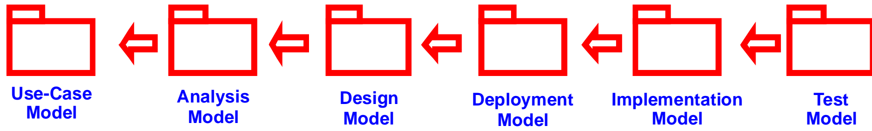


Figure 2.10. The primary model set of the Unified Iterative-Incremental Software Development Process [4].

Models provide following issues:
- System abstraction,
- Different perspectives on the system,
- Relationships between models.

The definition of the *Process* is as follows:
- The process as the software development process is a complete set of activities needed to map user requirements into a set of artefacts that represent software development factors:
  - Organizational,
  - Domain,
  - Life-cycle,
  - Technical.

Software *Tools* allow to:
- Automate the process,
- Standardize of process and product,
- Support the entire software lifecycle: defining of requirements, visual modelling and design, programming, testing.

2.4. The role of the Unified Modelling Language - UML

UML is the language supporting the iterative - incremental unified process of the software development. UML can be used for the software development by defining the UML diagrams. Diagrams represent the structure and behaviour of the software [4], [6].

UML diagrams for modelling structure are as follows:
- Package Diagrams,
- Class diagrams,
- Object Diagrams,
- Mixed Diagrams,
- Component diagrams,
- Deployment diagrams.

UML diagrams for modelling behaviour are as follows:
- Use-case diagrams,
- Activity Diagrams,
- State diagrams,
- Communication Diagrams,
- Sequence Diagrams,
- Timing Diagrams,

– Interaction Diagrams.

There are many benefits of using UML 2:

– Teamwork,
– Overcome the complexity of the project,
– A formal, precise presentation of the project,
– Creating a standard project,
– The opportunity of testing the software in an early stage in its development.

3. Overview of design patterns for supporting information systems modelling

In software engineering, emphasis is placed on software complexity and performance. They determine the quality of software. Another issue is efficient management of the software development. It is easier to manage the software development when using the principles of building software. One of them is the use of design patterns. There were doubts whether the use of design patterns supports the creation of high-quality software in an efficient manner. This introduction explores some of the issues surrounding design patterns and considers arguments from both the supporters and adversaries.

On one side of discussion are the advantages of applying design patterns, which result in an increase in software quality (such as completeness of abstraction, reusability, understandability, maintainability, testability), estimated cost of production, functionality offered and improvement of the project management. On the other side of the discussion there are groups of programmers who oppose the advocates of design patterns because these groups fear a deterioration of software performance and the difficulties in the implementation of design patterns in some languages.

A design pattern is a general reusable solution to commonly occurring problems in the software design and it is not a finished design. Therefore, it cannot be transformed directly into a code. Patterns as a kind of the description or the template provide a way to solve a problem in different situations. They typically show relationships between classes and interactions among the objects that are related but they do not provide the final specification of application classes and objects. However, design patterns facilitate the specification of the application classes and objects.

Therefore, efforts have been made to codify design patterns in creating multi-tiered software. Developers should have the knowledge of techniques based on design patterns which are used in object-oriented analysis, design and programming. They will develop their analytic skills for building object oriented multi-tiers software based on design patterns of the Client, Web, Business and Enterprise Information System Tiers.

The use of design patterns improves the management of software development. This follows from the fact that design patterns are composed of several sections. On the one side, there is the classification based on the Structure, Creational, and Behavioural sections [3], [6], [8] and on the other side there is classification supporting design and implementation of the multi-tier software such as Presentation, Business and Integration Patterns [2].

These sections describe *a design motif*: a prototypical *micro-architecture* that developers introduce to their particular designs, based on well-defined rules of applying design patterns. A micro-architecture as a design pattern is a set of program constituents (e.g. classes, methods...) and their relationships. As participants of the design group, developers well understand structure and organization of developed software if they are similar to the chosen design motif. Additionally, patterns allow developers to work using well-known, well-understood names of software activities. This improves the communication and organization of the workflow among the contractors of the project, such as analysts, designers, programmers and testers, and a project manager. They use the well-written documentation for a design pattern, which describes the context, in which the pattern is used, the forces within the context that the pattern seeks to resolve, and the suggested solution.

In addition to this, design patterns can be improved over time by making refactoring, turning some micro-architectures onto functionality equivalent ones. Therefore, they prevent subtle issues that can cause a major problem, for example reduce the scalability and

performance of software. This manner of the software development makes it is more robust than management of ad-hoc designs.

However, some group of software engineering experts questions these claims. They diminish the importance of the role of design patterns in production software, and they even mention the negative impact on software development.

The one of them is necessity of workarounds for missing language features. It is discussed in many design patterns as workarounds for the limitations of languages such as C++ and Java [7]. For instance, the main goal of the Visitor pattern is to add new operations to existing classes without modifying them. It is implemented in such languages as C++ and Java as a class, which is declared as a syntactic structure with a specific and closed set of methods. These patterns need not be implemented in a language that supports multi-methods as Common Lisp, because the new methods can be added outside of the class structure without modifying its structure. Similarly, in C++ and Java the Decorator pattern carries out functions of dynamic delegation, as found in Common Lisp, Objective C, Self and JavaScript. Many patterns only imply object orientation such as the Iterator pattern as a generalization of 'for' loops, an equivalent notion of loop.

The other criticism of design patterns is that in order to achieve flexibility, design patterns usually introduce additional levels of indirection. In some cases these levels may complicate the resulting designs and decrease application performance.

In addition, a pattern must be always programmed into each application that uses it because of the definition of design patterns. Therefore some authors believe that use of design patterns is a step backward from software reuse as provided by components. Some experts are working on transformation of design patterns into the functional-oriented reusable components.

In conclusion, the use of design patterns could lead to many benefits for developers, but could also lead to lower quality software. Therefore, developers should hone their skills on analysis, design and programming of multi-tiers applications based on design patterns fulfilling the user requirements for functionality, quality and cost of software.

In the Section 3.1, fundamentals of identification of design patterns are presented. The Sections 3.2- 3.4 present patterns as follows: creational, structural and behavioural.

3.1. Fundamentals of the design patterns identification

There are some issues tied with design patterns:

- A well-built object system is full of object-oriented patterns,
- A pattern is usually adopted to solve a typical problem in the given context,
- A structure of a design pattern is represented in the form of class diagram,
- The pattern behaviour is showing up using the sequence diagram,
- The design pattern represents the relationship between problem and solutions (according to G. Booch, J. Rumbaugh, I. Jacobson, UML User's Guide).

The main experts of software engineering represent the significant opinions upon the design patterns:

- Each pattern consists of three parts, which express the relationship between a particular context, problem and solution (based on Christopher Alexander),
- Each design pattern is a three-part rule that expresses the relationship between a particular context, a recurrent distribution of forces in this context, and software configuration, allowing for the mutual balancing of these forces in order to solve the task (based on Richard Gabriel),

– A design pattern is an idea that proved to be useful in a real context and will probably be useful in another (based on Martin Fowler).

These opinions support the ways of identification the design patterns during software development and ability of reuse some known patterns in this process. As one kind of classification there are three types of design patterns: creational, structural and behavioural. They are based on concepts of aggregation, inheritance, interface and polymorphism. In other words, they represent the most successful of paradigms of object- oriented methodology in structure and behaviour of software.

Each pattern is described by using the following template:

– A solved problem,
– A solution of a problem with description of components of a pattern,
– A client of pattern,
– A result.

3.2. Creational patterns

The main goal based of using creational patterns in software is isolation of rules for creating objects from rules that determine how the use of these objects (separation of the code for creating objects from the code that uses objects).

The list of creational patterns and aspects, which can change, is as follows [3], [6], [8]:

1. Builder - a way of creating of the complex objects.
2. Abstract Factory - a family of objects.
3. Factory Method - a subclass of the created object.
4. Prototype - a class of the created object.
5. Singleton - only one copy of the class.

3.2.1. Abstract Factory

Problem: Creating the appropriate families of related or dependent objects in the following cases:

– Different operating systems,
– Different requirements for effectiveness, efficiency, etc.,
– Various versions,
– Different types of co-application (such as different types of databases),
– Different functionality for different users,
– Different groups of elements depending on the settings related to the location (e.g. data format).

Solution: A *Client* object uses the *AbstractFactory* interface and the *AbstractProduct* interface. The *ConcreteProduct* and *ConcreteFactory* are classes that implement these interfaces. Each *ConcreteFactory* object can create one of the families of *ConcreteProduct* objects (Figure 3.1).

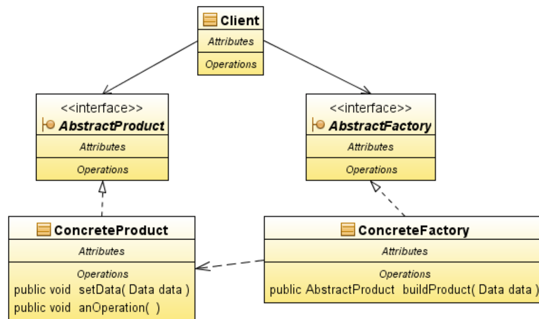


Figure 3.1. Class Diagram of the *Abstract Factory* Pattern.

Client: The *Client* object manages creation of objects by the factory, but it is independent of the rules on the creation of these objects (Figure 3.2).

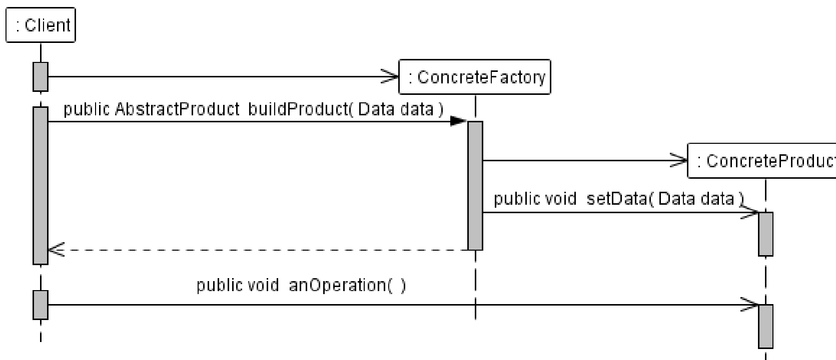


Figure 3.2. Sequence diagram of creation of the *ConcreteProduct* object of the *Abstract Factory* Pattern.

Result:

– Isolation of rules for creating objects from rules that determine how the use objects,
– To define the rules to create objects, that can best achieve the main goals of the application,
– Configuring applications using associated objects,
– The system uses created objects, knowing only their base classes.

3.2.2. Builder

Problem: Creating a complex custom objects represented in different ways.

Solution: The *Director* object has a request to the *ConcreteBuilder* object that implements the *Builder* interface, for creating the *Product* object (Figure 3.3).

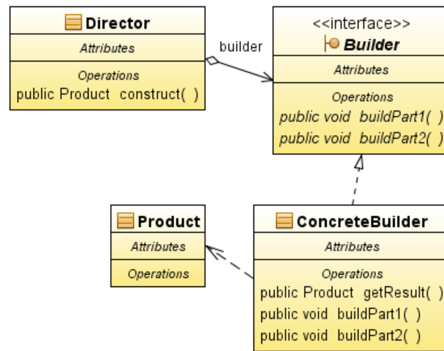


Figure 3.3. Class Diagram of the *Builder* Pattern.

Client: A *Client* instructs the *Director* object to create a *Product* object using the supplied the *ConcreteBuilder* object (Figure 3.4).

Result: The *Director* objects receive the *Builder* abstract interface that allows you to construct *Product* objects freely.

The pattern supports separating code used to construct *Product* objects from code using these objects. For example, the Director objects are created to read documents in RTF or XML format using the *ConcreteBuilder* as the ASCII converter, which supplies the ASCII format document represented by the *Products* objects, from the documents of the RTF or the XML formats. This does not affect of the client code, because of converting abilities of many formats of different *ConcreteBuilder* objects.

The algorithm to build a *Product* object is created independently of its components that may be of any type. What we achieve is better control of the construction of the *Product* object by using operations, implemented through an interface *Builder* by the *ConcreteBuilder* object and controlled by the *Director* object.

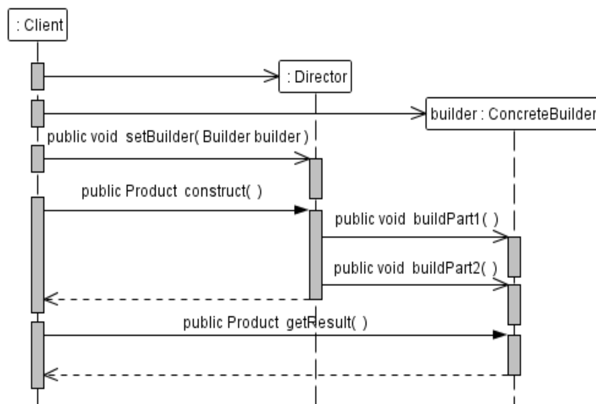


Figure 3.4. Sequence Diagram of creation of the *Product* object of the *Builder* Pattern.

3.2.3. Factory Method

Problem: This pattern defines an interface for creating objects, and its subclasses are allowed to decide which class is to be each of these objects.

Solution: The *Creator* interface declares the factory method, which in turn declares the produced object, derived from the *Product* object. The action is transferred to the *ConcreteCreator* object, which produces the *ConcreteProduct* object by means of the factory method (Figure 3.5).

Client: The client selects the *ConcreteCreator* object with the appropriate factory method, which creates the *ConcreteProduct* object. Processing the *ConcreteProduct* object by the *ConcreteCreator* lies in the fact that only a factory method knows the representation of the object and manner of its creation, while other methods know interface of the *Product* abstract class and should only use methods of this interface (Figure 3.6).

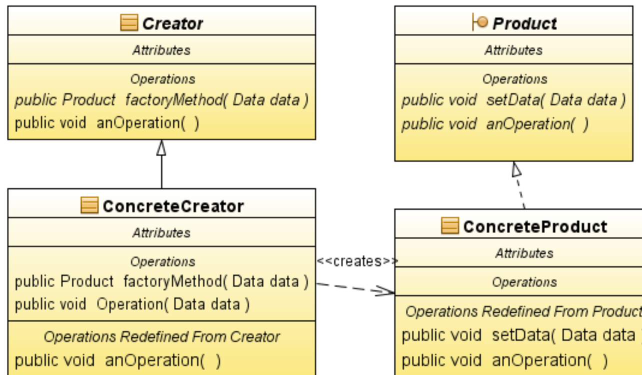


Figure 3.5. Class Diagram of the *Factory Method* Pattern.

Result:

– Isolation of rules for creating objects from rules defining how to use the objects under the family implementing the *Creator* interface,
– To define the rules to create objects that can best achieve the objectives of an application,
– The *ConcreteCreator* object should serve well to use the *ConcreteProduct* object or its derivative from the *ConcreteProduct* object, but only its factory method should know the rules for creating *ConcreteProduct* objects.

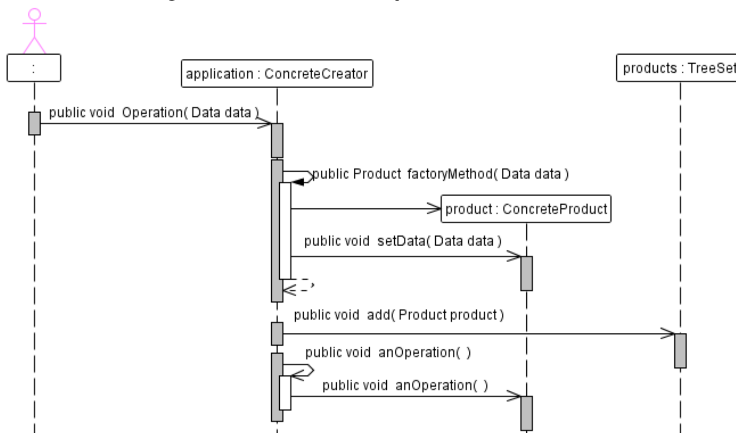


Figure 3.6. Sequence Diagram of creation and using the *Product* object of the *Factory Method* Pattern.

3.2.4. Prototype

Problem: Separating code to create objects from code for using them, without building a class hierarchy of factories in a situation where you need a limited number of created objects.

Solution: The *Client* object receives the needed object with the *Prototype* interface as the *ConcretePrototype* object by cloning objects (Figure 3.7).

Client: The *Client* object uses the cloned objects as the *ConcretePrototype* objects, which implement the *Prototype* interface (Figure 3.8).

Result:

– Adding and deleting objects without using any object factory,
– Reducing the number of classes,
– Dynamic loading *ConcretePrototype* classes implementing the *Prototype* interface.

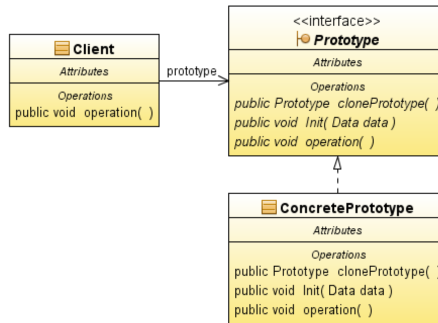


Figure 3.7. Class Diagram of the *Prototype* Pattern.

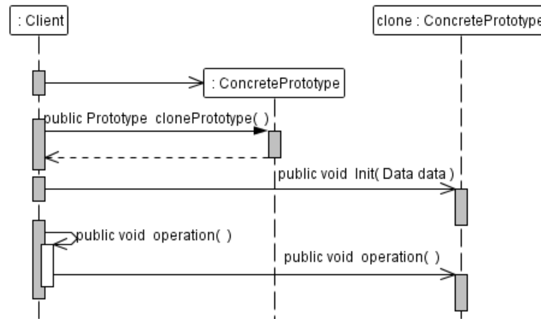


Figure 3.8. Sequence Diagram of the *Prototype* Pattern.

Below, it is an example of using the method *clone,* implemented the *Prototype* Pattern in the *ArrayList* class.

```
public Object clone() {      // the clone method of the ArrayList class
 try{
        ArrayList<E> v = (ArrayList<E>) super.clone();
        v.elementData  = Array.copyOf(elementData, size);
        v.modCount = 0;
        return v;
      } catch (CloneNotSupportedException e)
        { throw new InternalError(); }
 }
public class Main {        //the example of the use of  the Prototype pattern
 public static void main (String[] args) {
    ArrayList coll1, coll2 = new ArrayList();
    coll2.add(new Integer(1));
    coll2.add("B");
    coll1 = (ArrayList) coll2.clone();
    coll1.add("C");                  // [1, B, C]
    coll2.remove(0);    }              // [B]
}
```

3.2.5. Singleton

Problem: A guarantee, that only one instance of a class exists in software. There is a global access to this object such as a file system or window system.

Solution: A *Singleton* object makes sure on its own that no other object was constructed of the same type (Figure 3.9).

Client: The *Singleton* object can have multiple clients.

Result:

– Reduced name space,

– Controlled access to the single copy.

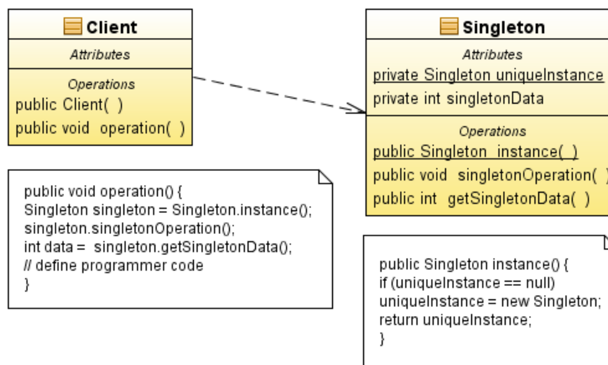


Figure 3.9. Diagram Class of the *Singleton* Pattern.

3.3. Structural patterns

The main goal of structural patterns is to make classes and objects into larger structures.

There are two kinds of structural patterns. On one side, there is a kind of a class design pattern, which uses the inheritance and polymorphism to make the structures of interfaces and their implementations. On other side, there is the object design pattern which describe the way how to combine objects in order to obtain new functionality, even during execution of a program.

There is the list of the structural patterns and aspects, which can change, as follows [3], [6], [8]:

1. Adapter - class and object design pattern; an interface of an object.
2. Bridge - object design pattern; an implementation of the object.
3. Composite - object design pattern; a structure and a scheme of the object.
4. Decorator – object design pattern; an obligation of the object without defining of the subclass.
5. Facade - object design pattern; an interface of a subsystem.
6. Flyweight - object design pattern; a cost of storing objects.
7. Proxy - object design pattern; a way of an access to the object; its location.

3.3.1. Adapter

Problem: The class interface should be adapted to the interface expected by the client, e.g. the pattern associated with the change of class libraries that support graphics.

Solution: The *Client* object uses the *Adapter* object, which implements the *Target* interface, and mediates the access to methods of the *Adaptee* objects (Figure 3.10, Figure 3.11).

Client: The *Client* object is independent of changes in methods or headers of their definition of a library class (*Adaptee*) that implement specialized operations such as graphical operations, because the client always uses methods of the *Adapter* object, which does not change the headers of the methods (Figure 3.10, Figure 3.11).

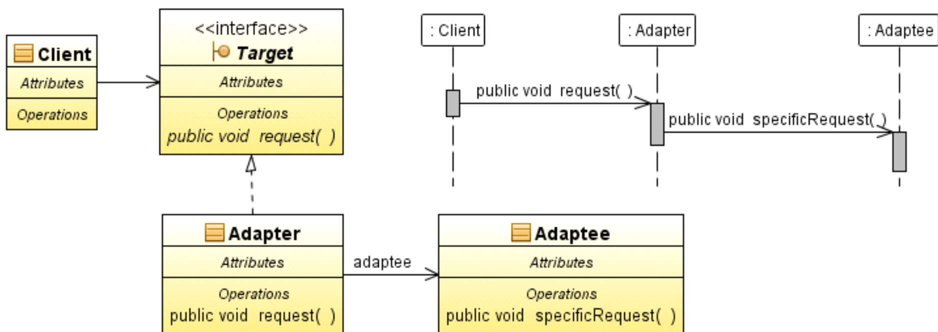


Figure 3.10. Class and sequence diagrams of an object pattern type of the *Adapter* Pattern.

Result:

– *Adapter* object (making objects):
    - It allows one *Adapter* object to collaborate with multiple objects such the *Adaptee* object and its derivatives. In the *Adapter* object can add new functionality – it is an advantage of the *Adapter* pattern.

- In case of a hierarchy of the *Adaptee* classes, which reflect the change in the behaviour of this object, the *Adapter* object must refer to the subclasses of *Adaptee* object instead of the *Adaptee* type – it is a defect of *Adapter* pattern.

– *Adapter* classes (multiple inheritance):
  - Pattern adapts the interface of the class, which is used in the program to the interface of the new class libraries, but does not apply to the subclasses – it is the defect of *Adapter* pattern.
  - The *Adapter* object allows you to redefine the behaviour of the *Adaptee* class because of its subclass– it is an advantage of *Adapter* pattern.

It introduces only one *Adapter* object that provides the adaptation of the one *Adaptee* object – it is an advantage of the *Adapter* pattern.

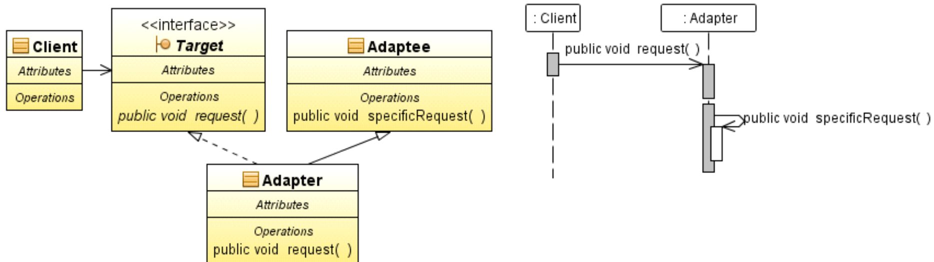


Figure 3.11. Class and sequence diagrams of the class pattern type of the *Adapter* Pattern.

3.3.2. Bridge

Problem: You should separate the abstract from the implementation, so that they can change independently of one another.

Solution: The *Abstraction* interface is implemented by the *RedefineAbstraction* class. The *RedefineAbstraction* objects have methods that use the methods of the *Implementor* abstract class (or an interface), which are implemented by specific *ConcreteImplementor* classes, cooperating with various classes (with different interfaces) from a variety of platforms and libraries (Figure 3.12, Figure 3.13).

Client: The client uniformly treats each *RedefineAbstraction* object without committing to a specific platform or a library (Figure 3.14).

Result:

– Separation of abstraction from implementation to eliminate dependencies while compiling a program or activity,
– Introduction of a multi-tier architecture,
– Extensibility of class hierarchies of the *Abstraction* and *Implementor* classes,
– Easy addition of new objects,
– Hiding implementation details.

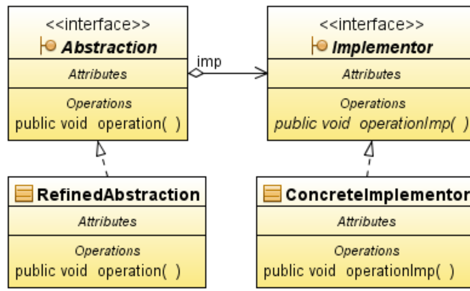


Figure 3.12. Class Diagram of the *Bridge* Pattern.

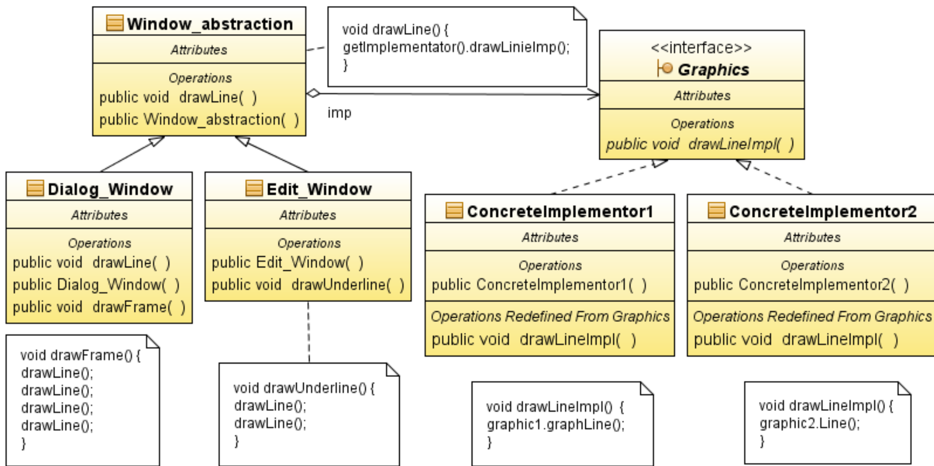


Figure 3.13. Class Diagram of an example of the use of the *Bridge* Pattern supporting two kinds of frames of the GUI.

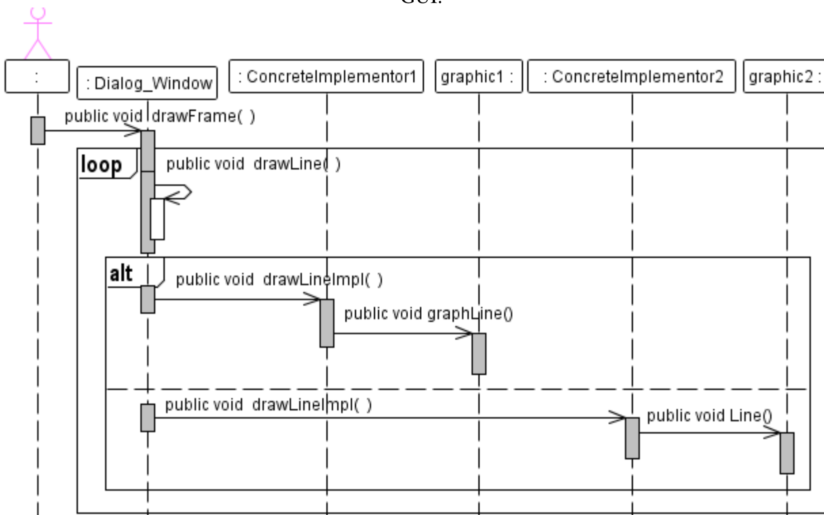


Figure 3.14. Sequence diagram of an example of the use of the *Bridge* Pattern supporting two kinds of frames of the GUI.

3.3.3. Composite

Problem: It combines objects in object-oriented data structures (tree) as a part-whole.

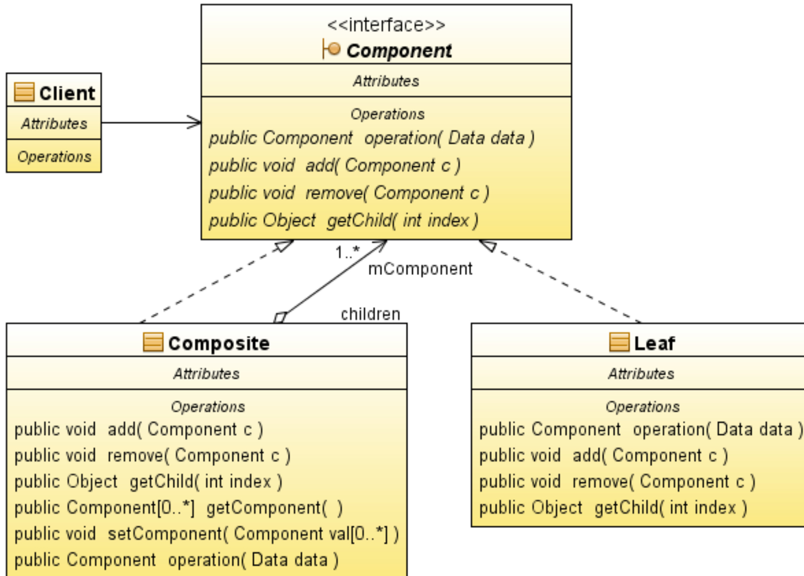


Figure 3.15. Class Diagram of the *Composite* Pattern.

Solution: The *Component* abstract class (or an interface) declares basic operations for graphical objects. The Composite object contains two sets of objects: *Composite* objects and containing *Leaf* objects, which in turn do not contain any objects at all. The *Leaf* objects implement the *Component* class (Figure 3.15, Figure 3.16).

Client: The *Client* object uniformly treats each element of the object structure - as objects of the *Component* type (Figure 3.17).

Result:

– Recursively grouping objects primary (type-*Leaf*) and complex objects (*Composite*),
– A simple construction of a client who does not need to distinguish between simple and composite objects,
– Easy addition of new facilities,
– Difficulties in maintaining restrictions of complex objects.

Implementation: a new class of "Boundry" such as *Swing* package (Figure 3.18):

– The *JComponent* class represents the *Component* class,
– The *JButton* class represents the *Leaf* class and the,
– The *JPanel* class represents the *Composite* class.

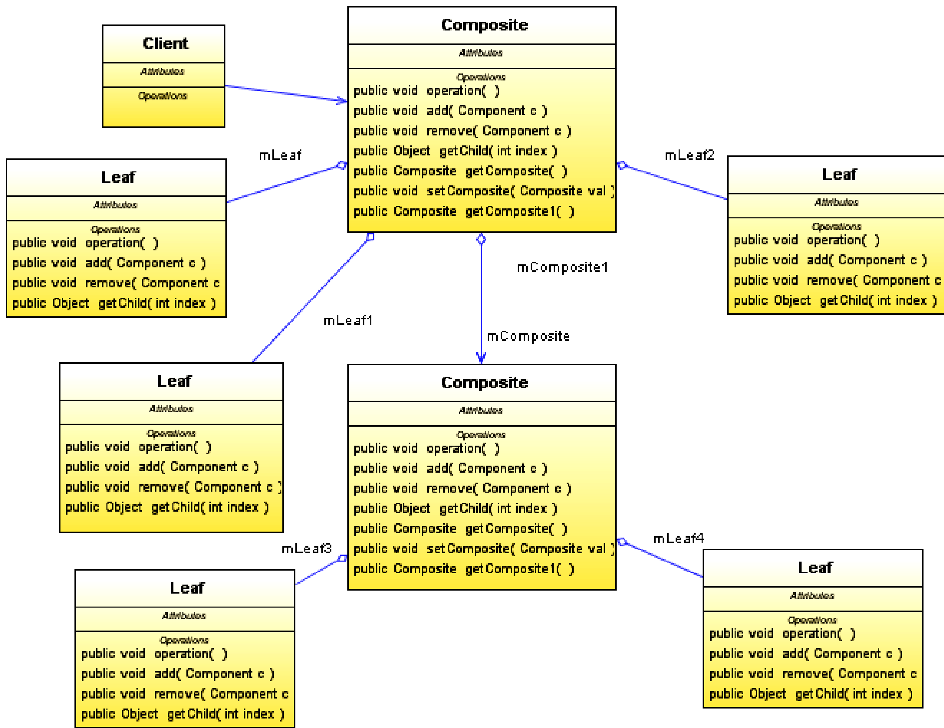


Figure 3.16. Object Diagram of the *Composite* Pattern.

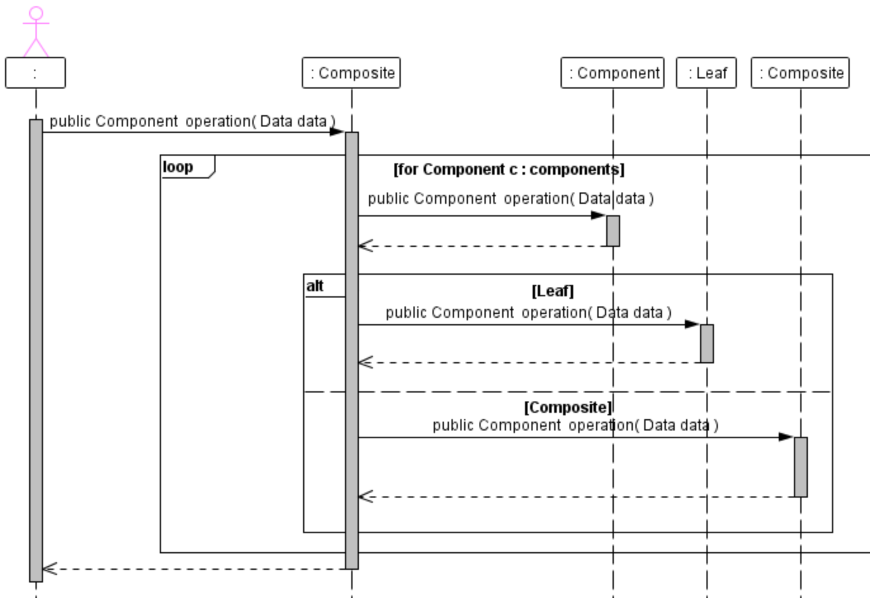


Figure 3.17. Sequence diagram of the key method of the *Composite* Pattern.

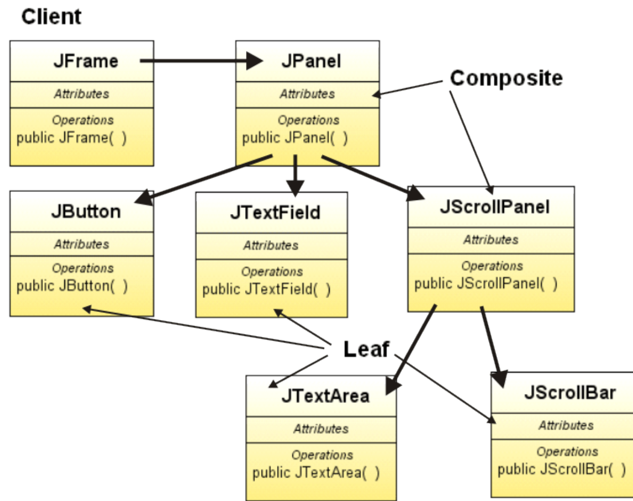


Figure 3.18. Example of an Object Diagram of the *Composite* Pattern as the GUI of the Swing library.

3.3.4. Decorator

Problem: Dynamically developing functionality of a facility as an alternative to creating a deep hierarchy of classes.

Solution: A *Component* is an abstract class (or an interface) for visual objects. Its interface defines operations of drawing and event handling implemented by the *ConcreteComponent* class. The *Decorator* abstract class (or an interface) inherits operations from the *Component* class (or an interface) and defines additional operations performed by the *ConcreteDecorator* object (Figure 3.19, Figure 3.20).

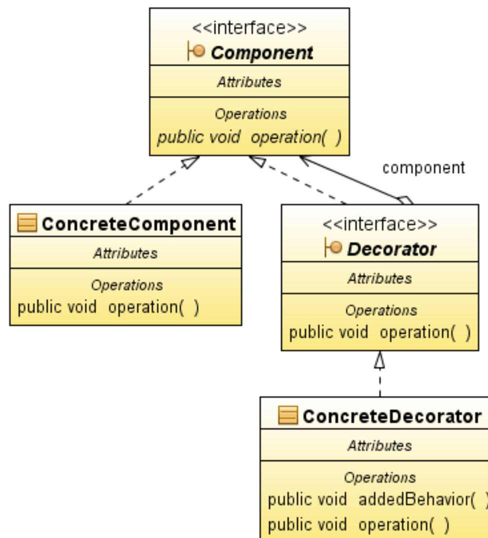


Figure 3.19. Class Diagram of the *Decorator* Pattern.

Client: The object executes with objects inherited from the *Component* class, with and without decorators (Figure 3.21, Figure 3.22).

Result:

- A dynamic and transparent solution to add additional components to the basic components,
- Easy removal of any additional functionality,
- Replacement of a class hierarchy containing the equivalent functionality on a permanent basis, by dynamically adding decorators with different functionalities.

Implementation: a new class of "Boundry" such as Swing package classes, library classes of Java Server Faces components.

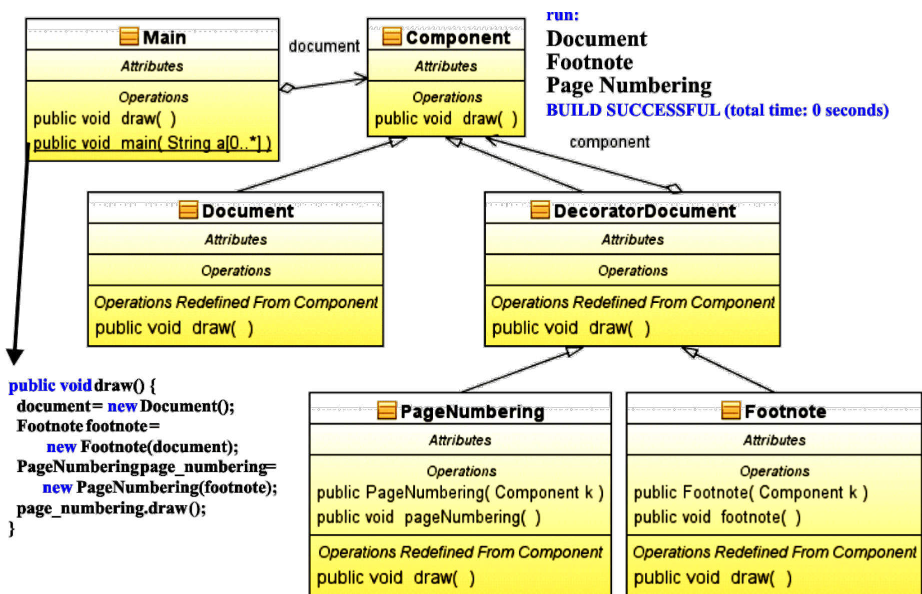


Figure 3.20. Class Diagram of an example of using the *Decorator* Pattern.

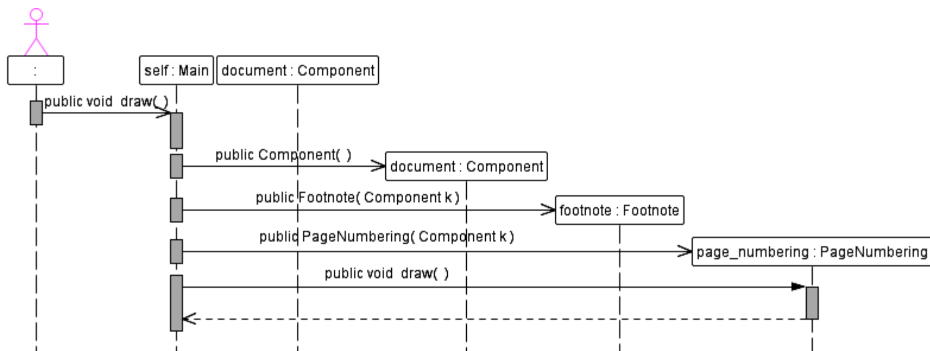


Figure 3.21. Sequence Diagram of the example of using the *Decorator* Pattern.

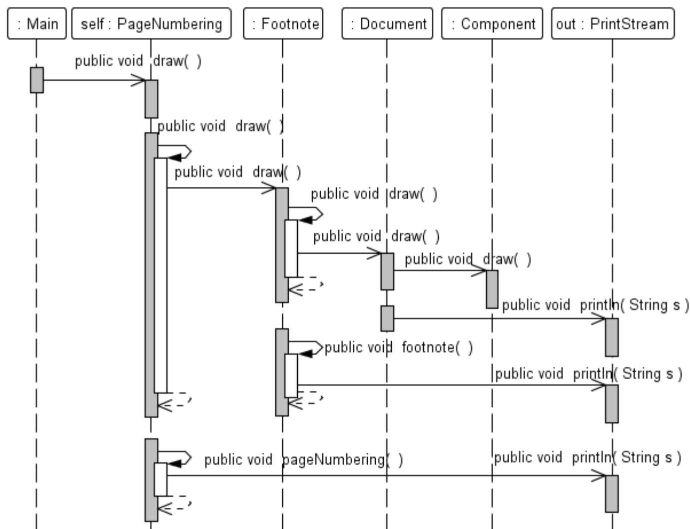


Figure 3.22. Sequence Diagram of the example of using the *Decorator* Pattern (related to Figure 3.20).

3.3.5. Façade

Problem: Grant only access to selected functions of the system tier (Figure 3.23).

Solution: It is an interface or interfaces of a system tier - facades provide several methods for selected groups of subsystems (Figure 3.24, Figure 3.25).

Client: receives only necessary methods (Figure 3.26).

Result:

– Release of important methods, only for example, a tier of the system use cases, hides classes of the system tier,

– The facade may prevent access to all methods of the encapsulated class.

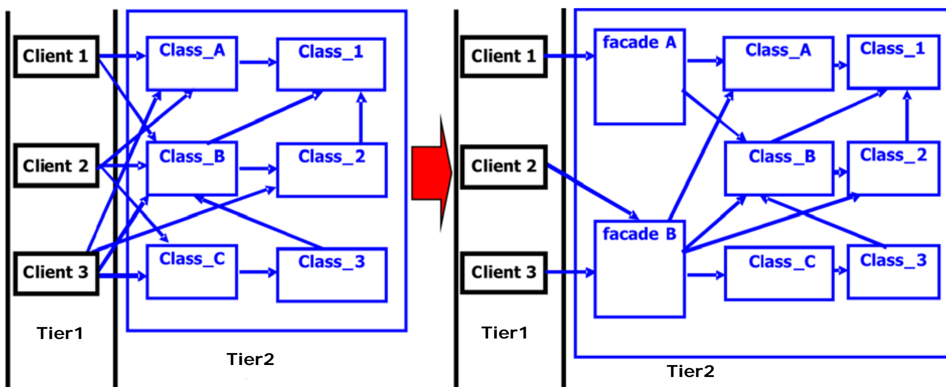


Figure 3.23. Refactorization of the Business Tier by using the *Façade* Pattern.

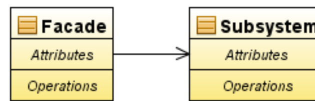


Figure 3.24. Class Diagram of the *Façade* Pattern.

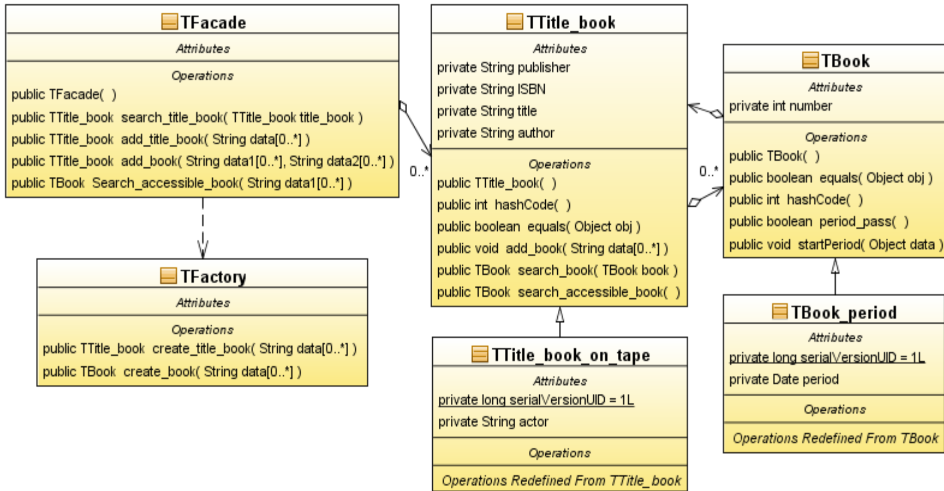


Figure 3.25. Class Diagram of an example of using the *Façade* Pattern.

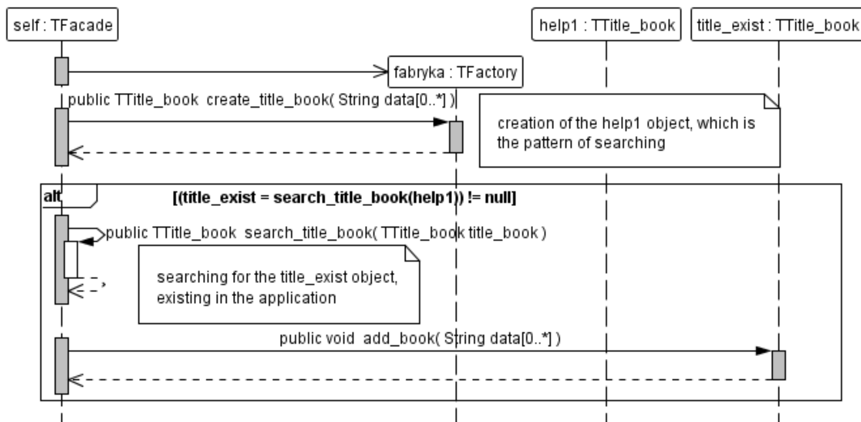


Figure 3.26. Sequence Diagram of an example of using the *Façade* Pattern (encapsulation and distribution of business logic between the *Façade* object and the *TTitle_book* of adding a new *TBook* object – by the *add_book* method of *the Façade* object and the *add_book* method of the *TTitle_book* object).

3.3.6. Flyweight

Problem: Repeated use of the same object - to share objects.

Solution: The *Flyweight* interface declares methods which are implemented by *ConcreteFlyweight* objects (shared use) and *UnsharedConcreteFlyweight* objects (used once) used by client applications. Objects-pollen are created and managed by the *FlyweightFactory* object (Figure 3.27).

Client: A client keeps references pools to flyweight objects.

Result: Memory savings by sharing facilities of „flyweight” objects,

Implementation (Figure 3.28, Figure 3.29, Figure 3.30), a new class of "*Boundry*" or "*Entity*", e.g.:

– Reference of the same object from a *TProduct1* family (flyweight) may be stored in many objects such as the *TItem* (client),
– Object References of the *TPromotion* object (flyweight) may be kept by one of the objects of the family *TProduct1* (client).

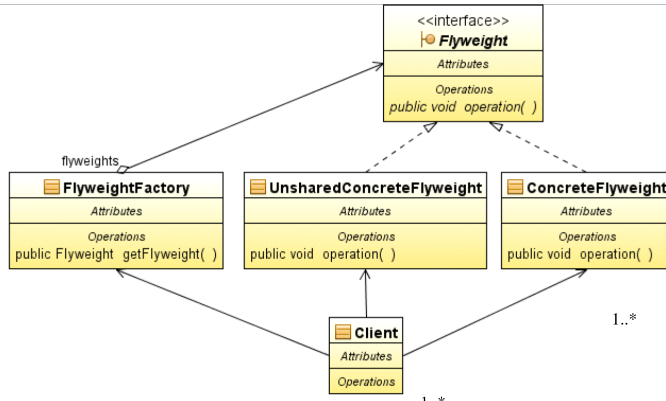


Figure 3.27. Class Diagram of the *Flyweight* Pattern.

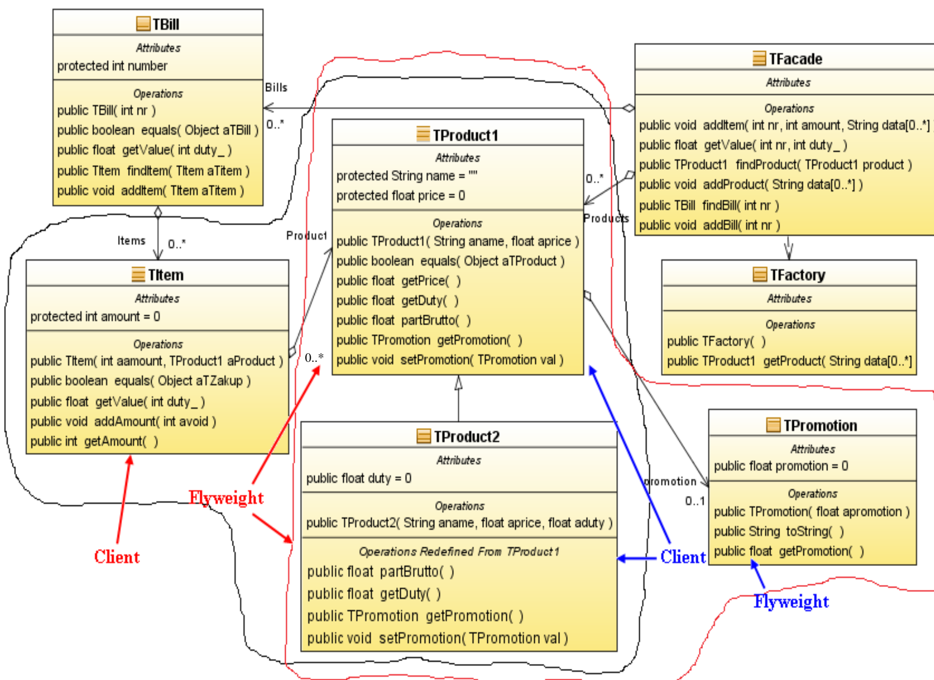


Figure 3.28. Class Diagram of an example of using the *Flyweight* Pattern.

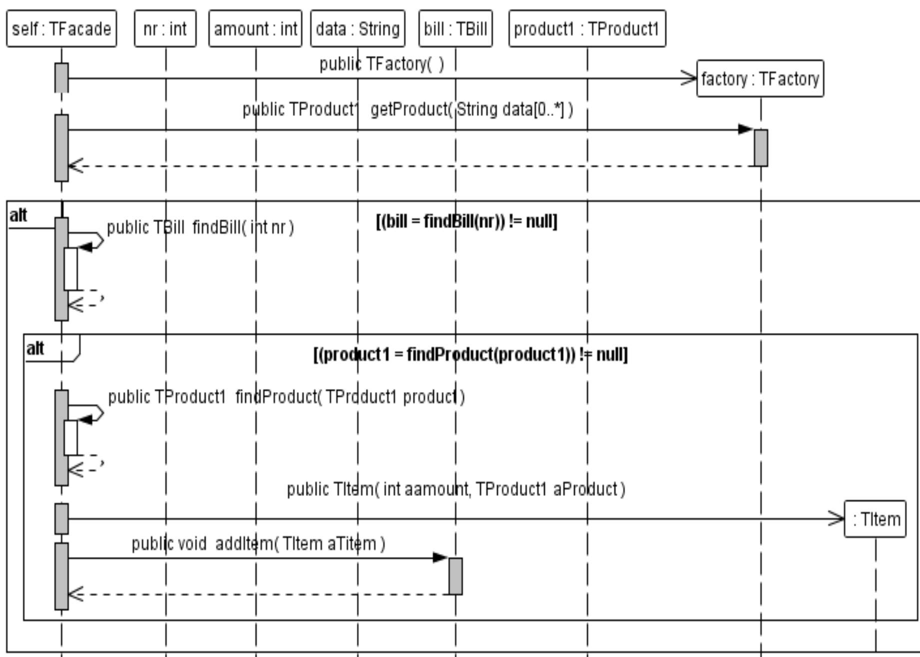


Figure 3.29. Sequence Diagram of an example of using the *Flyweight* Pattern for preparing bills (the *addItem* method of the *TFacade* class) – the family of *TProduct1* objects as the flyweights objects of *TItem* objects of *TBill* objects.

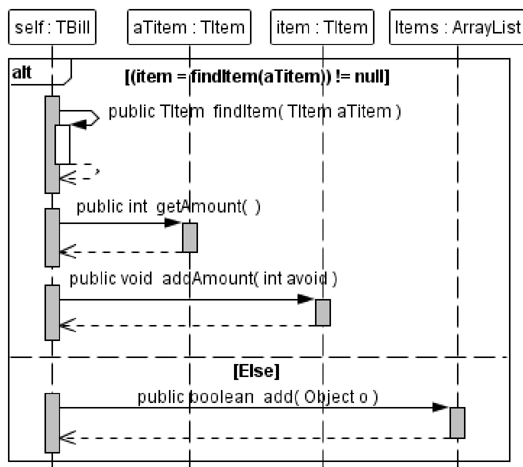


Figure 3.30. Sequence Diagram of an example of using the *Flyweight* Pattern for preparing bills (the *addItem method* of the *TBill* class) – the family of *TProduct1* objects as the flyweights objects of *TItem* objects of *TBill* objects (related to Figure 3.29).

3.3.7. Proxy

Problem: Representation one object to control access to default one.

Solution:

- A *Proxy* object stores a reference to the true object of the *RealObject* object and may replace the *RealObject* object because they have the same interface (a *Proxy* class inherits from the *Subject* class) and can control access to the *RealObject* object,
- The *Proxy* object may be a remote object, referring to the *RealObject* object, or provides the virtual access to the object by buffering the *RealObject* object, or prevents the access to the *RealObject* object by unauthenticated objects (Figure 3.31, Figure 3.32).

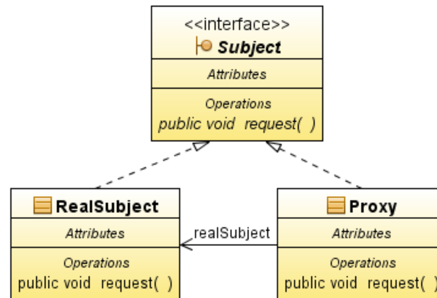


Figure 3.31. Class Diagram of the *Proxy* Pattern.

Client: Any object of any tier of the multitier systems (Figure 3.33).

Result:

- A remote *Proxy* object can hide *RealObject* objects in any address space,
- A virtual *Proxy* object improves performance by caching the data of the *RealObject* object and limits unnecessary operations on the *RealObject* object such as modifications of file records,
- The *Proxy* object provides security, because it can authorize or prohibit access to *RealObject* objects.

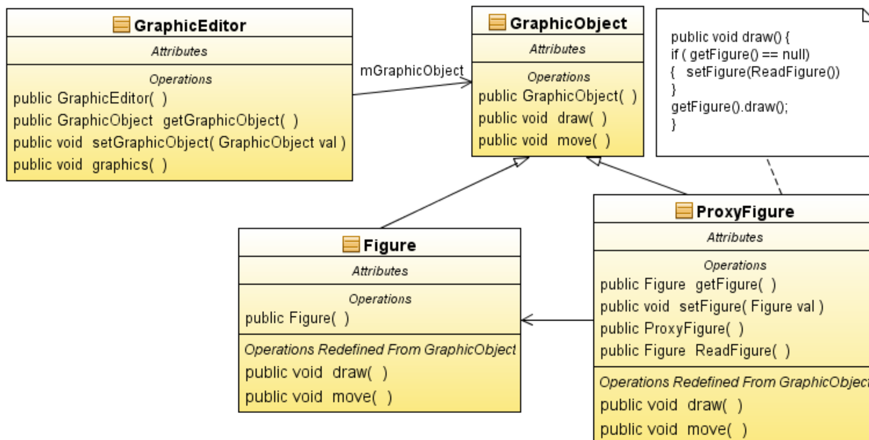


Figure 3.32. Class Diagram of an example of using of the *Proxy* Pattern.

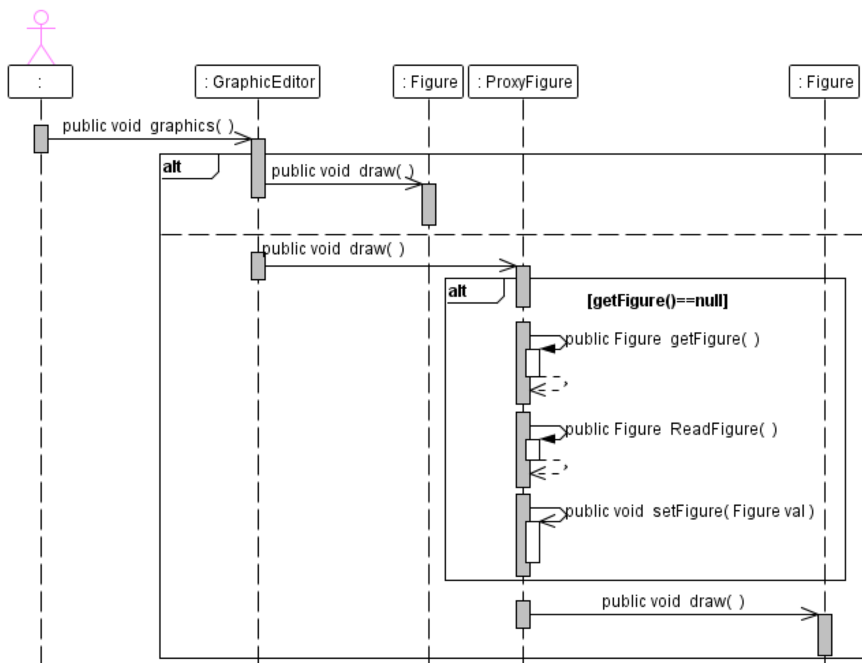


Figure 3.33. Sequence Diagram of an example of using of the *Proxy* Pattern (as the *ProxyFigure* object, which can also draw the figure such as the *Figure* object).

3.4. Behavioural patterns

The main goal of behaviour patterns allocation of algorithms and obligations, covering the patterns of objects and classes, and communication between objects.

The list of behavioural patterns and aspects, which can change, is as follows [3], [6], [8]:

1. Chain of Responsibility - an object, that can achieve a request.
2. Command - a condition and a way of realization of a request.
3. Interpreter - a grammar and an interpreter of a language.
4. Iterator - a way of the access and walking through elements of an aggregate.
5. Mediator - why and which objects influence one another.
6. Memento - which and when private information is stored outside the object.
7. Observer - number of objects, which are dependent on other objects and how dependent objects keep the actual state.
8. State - a state of objects.
9. Strategy - an algorithm.
10. Template method - steps of the algorithm.
11. Visitor - operations, which can be applied to objects without changing their classes.

3.4.1. Chain of Responsibility

Problem: The pattern creates a chain receiving objects and passes the request along a chain of objects, until any object, which can handle it.

Solution: The *Handler* interface declares the service of demands and possibly a reference to the successor. The *ConcreteHandler* object is responsible for claims, which are detected and supported (Figure 3.34).

Client: generates and directs requests to the list of *ConcreteHandler* objects (Figure 3.35).

Result: The object to be handled and the *ConcreteHandler* objects have no explicit knowledge about them and do not necessarily know the structure of the chain. A chain of commitments increases flexibility in allocating service requests by changing the subclasses of objects and structures of objects chain - but no guarantee of receiving a request.

Implementation: Use of the event handlers.

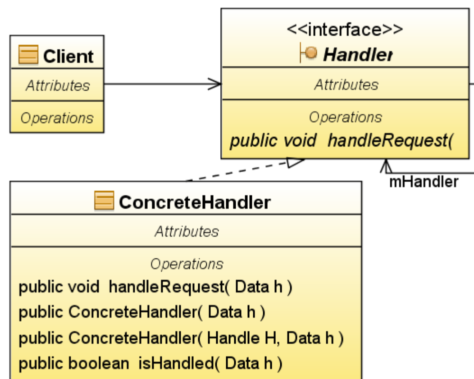


Figure 3.34. Class Diagram of the *Chain of Responsibility* Pattern.

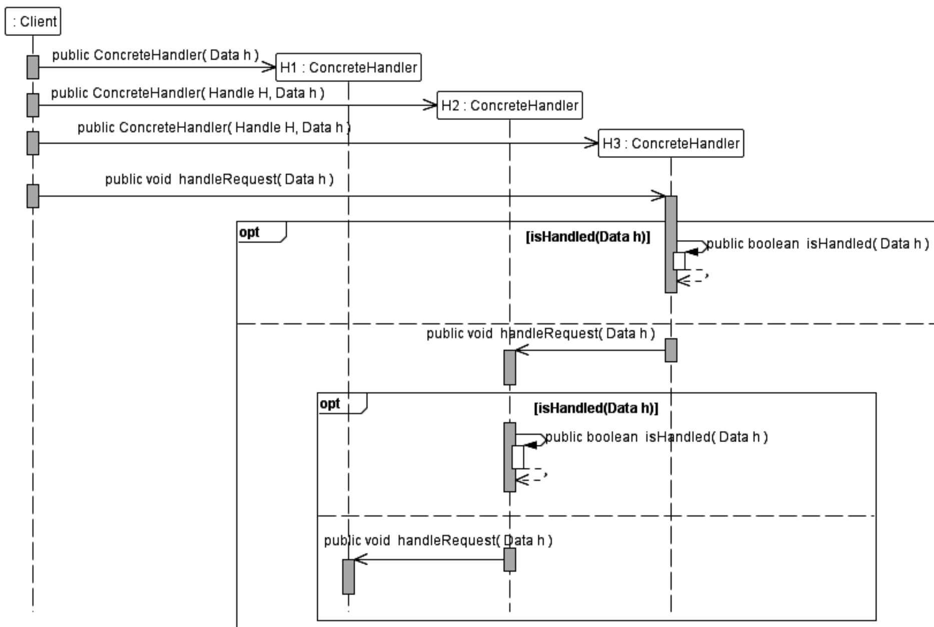


Figure 3.35. Sequence Diagram of the *Chain of Responsibility* Pattern.

3.4.2. Command

Problem: Encapsulation of requests in the form of an object, allowing to parameterize the various demands of clients and records requests.

Solution: The *Command* interface declares the performed operations. The *ConcreteCommand* object defines the link with the *Receiver* object and its action, and implements the execute method by calling the methods of the *Receiver* object. The *Invoker* object asks the *ConcreteCommand* object to perform request. The *Receiver* object knows how to execute a request (Figure 3.36).

Client: The *Client* creates the *ConcreteCommand* object and sets *Receiver* objects, which perform the request, and the caller such as the *Invoker* object (Figure 3.37).

Result (Figure 3.37):

- Separation of the objects which are triggering the operation and which are executing these operations,
- Ability to create complex objects with the *ConcreteCommand* objects,
- Ease of insertion of new classes of the derivative-type *Command.*

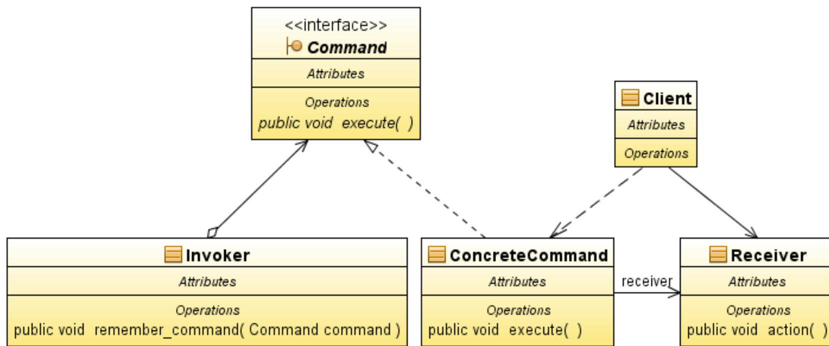


Figure 3.36. Class Diagram of the *Command* Pattern.

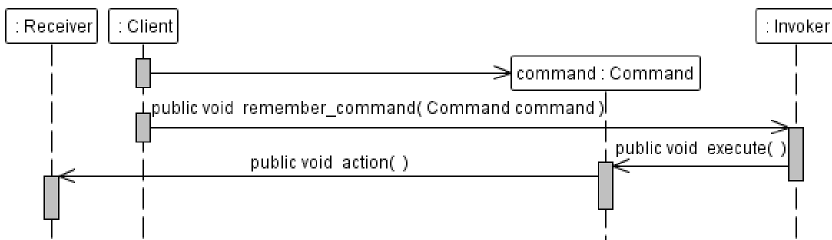


Figure 3.37. Sequence Diagram of the *Command* Pattern.

3.4.3. Interpreter

Problem: A definition of representation for the grammar of the given language and the interpreter of sentences written in a language-defined by the grammar.

Solution: The *Context* object contains global information for the interpreter. The *Client* object gets the built the *AbstractSyntax* tree representing the sentence of the language and calls its interpret method - a tree made up of the *TerminalExpression* and

NonterminalExpression objects, which implement the *AbstractExpression* interface (Figure 3.38).

Client: The *Client* object builds or gets a syntax tree, and starts the process of interpretation of the sentence represented by the tree (Figure 3.39).

Result:

– Easy modification of the grammar,
– Easy to implement different grammar,
– Difficulties to handle complex grammar - each class represents at least the one production rule,
– Adding new ways of interpreting the expression by the modifications of the classes.

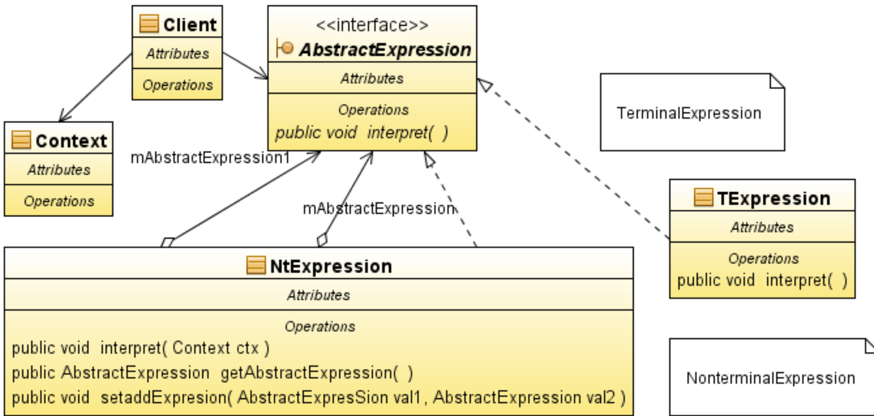


Figure 3.38. Class Diagram of the *Interpreter* Pattern.

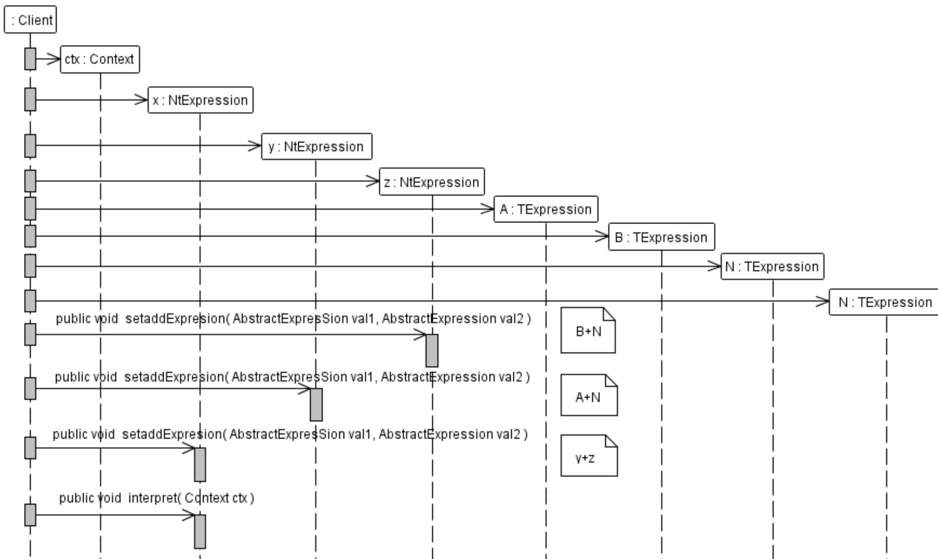


Figure 3.39. Sequence Diagram of the *Interpreter* Pattern.

Example:

TerminalExpression = {-6, -2, 1, 5, x, y},

NonterminalExpression = {S, A, AN, B, N},

and rules **BNF** notation:

<S> ::= <AN><AN>

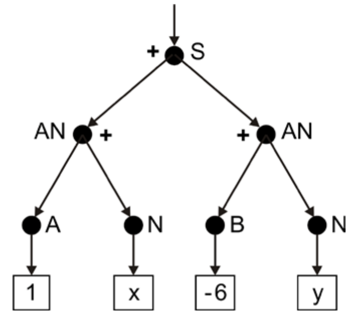
<AN> ::= <A><N>|<B><N>

<N> ::= x | y

<A> ::= 1 | -2

<B> ::= 5 | -6

((1+x)+(-6+y))



3.4.4. Iterator

Problem: Sequential, multiple and equal access to aggregated elements of the object without specifying the internal structure of the elements.

Solution: The Iterator interface defines access to elements of units and how to pass through the unit. The *ConcreteIterator* object implements the Iterator interface. The *Aggregate* declares the interface through passes the Iterator interface. This is the only a link between *ConcreteAggregate* and *ConcreteIterator* objects (Figure 3.40).

Client: The *Client* object can track in the *ConcreteAggregate* object by using the *ConcreteIterator* object, which an element is the current and next or previous (Figure 3.41).

Result:

- Opportunity of free passage through the *ConcreteIterator* object the *ConcreteIterator* interface simplifies of the *ConcreteAggregate* interface,
- At any given time, it can perform a lot to go through the *ConcreteIterator* object.

Implementation: a new class of "Control" for example, the *Iterator* and *ListIterator* classes in package java.util.

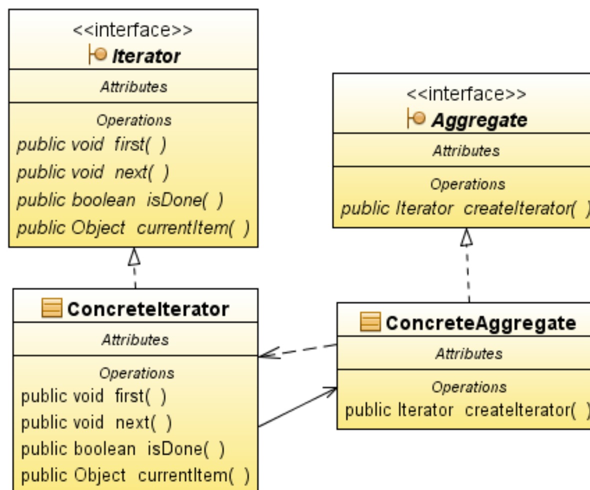


Figure 3.40. Class Diagram of the *Iterator* Pattern.

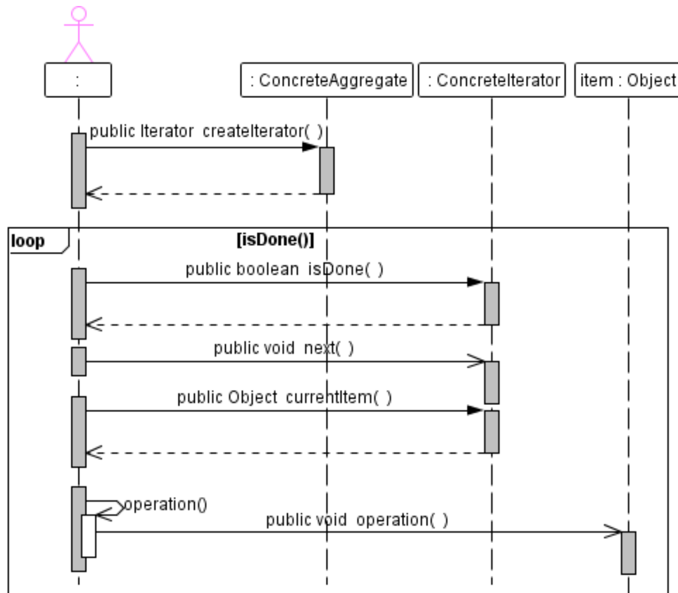


Figure 3.41. Sequence Diagram of the *Iterator* Pattern.

3.4.5. Mediator

Problem: The pattern allows you to limit a number of links between objects that interact in complex ways, and allows you to change the way they communicate.

Solution:

– The *Mediator* declares an interface in the agreement of the *Colleague* interface, so everyone *ConcreteColleague* object knows the operations of the *ConcreteMediator* object - each *ConcreteColleague* object need not communicate with another *ConcreteColleaque* object, but with a *ConcreteMediator* object; the *ConcreteMediator* object coordinates cooperation of many *ConcreteColleague* objects (Figure 3.42),

– The *ConcreteColleague* objects send requests to the *ConcreteMediator* object and then the *ConcreteMediator* object sends these requests to the appropriate *ConcreteColleague* objects (Figure 3.42).

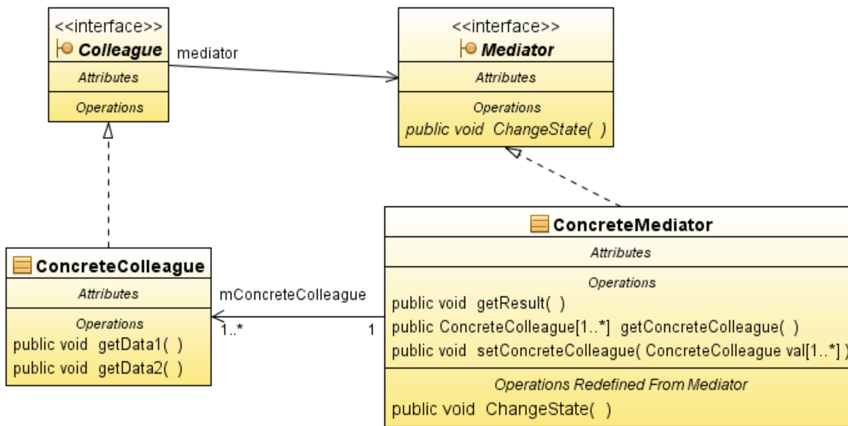


Figure 3.42. Class Diagram of the *Mediator* Pattern.

Result (Figure 3.43):

– The *ConcreteMediator* object gathers behaviour that would otherwise be placed in many *ConcreteColleague* objects,

– You can combine different types of object properties such the *ConcreteClleague* and the *ConcreteMediator*,

– Simplification of protocols to communicate through one-to-many associations between the objects such the *ConcreteMediator* and *ConcreteColleague*, replacing many of the objects derived from many *ConcreteColleague* objects,

– The generalization of the cooperation between objects, which implement the *Colleague* interface by the *Mediator* interface,

– The functionality of the objects, which implement the *Mediator* interface, can lead to very complex implementations, which will be difficult to maintain.

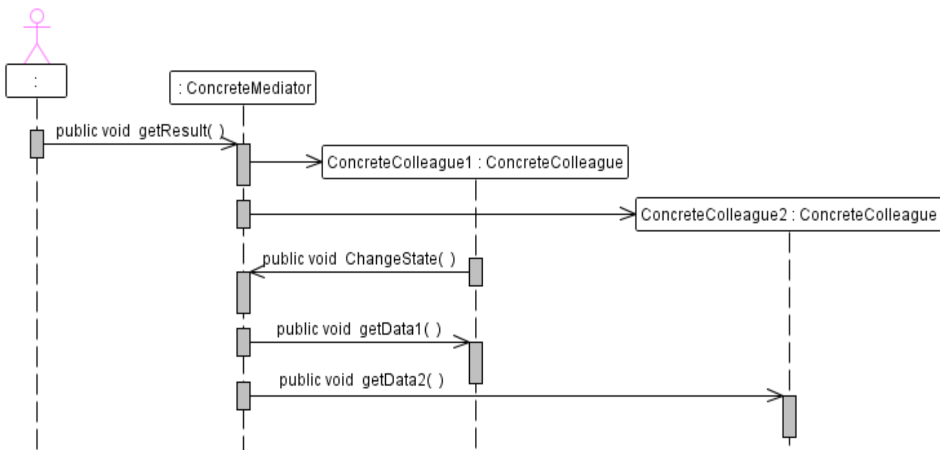


Figure 3.43. Sequence Diagram of the *Mediator* Pattern.

3.4.6. Memento

Problem: The pattern stores a state of some objects to restore this state at some later moment.

Solution: The *Caretaker* object is responsible for the care of the *Memento* object, but does not refer to methods of this object - may, however, give the *Memento* object to other objects. The *Originator* object creates the *Memento* object, which includes its state and uses it to restore its status - it has access to all methods of the *Memento* object. The *Memento* object holds the state of the *Originator* object (Figure 3.44).

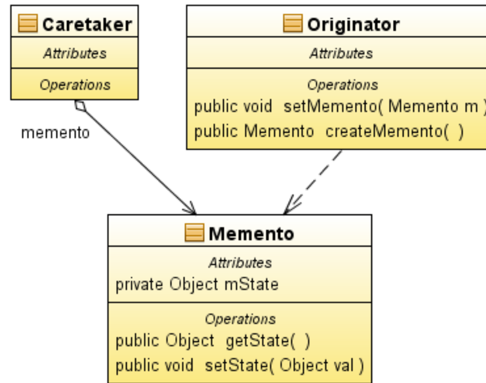


Figure 3.44. Class Diagram of the *Memento* Pattern.

Result (Figure 3.45):

– Maintaining boundaries of encapsulation, but posing difficulties in its maintenance,
– Simplification of the *Originator* object,
– Performance degradation,
– Difficulty of implementation,
– Difficulty in maintaining the *Memento* objects.

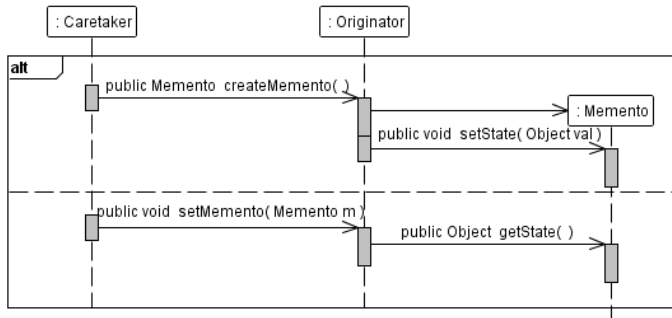


Figure 3.45. Sequence Diagram of the *Memento* Pattern.

3.4.7. Observer

Problem: Define the relationship one-to-many relationship between objects - when one object changes its state, all dependent objects are automatically notified and updated.

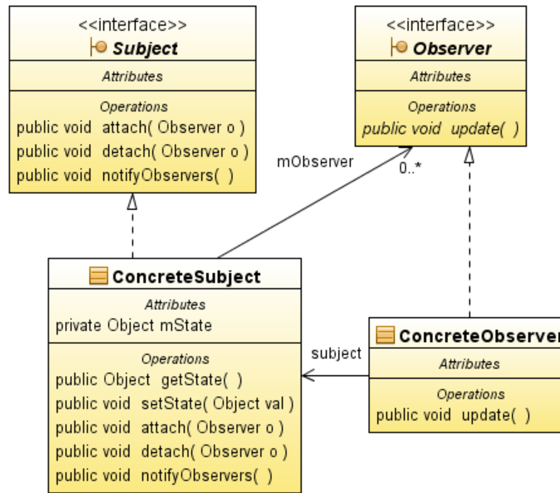


Figure 3.46. Class Diagram of the *Observer* Pattern.

Solution: The *Subject* interface knows its observers (*Observer* interface) and declares the attaching and detaching operations of observers. The *Observer* interface declares the operations of updating observed objects (Subject interface). The *ConcreteSubject* objects hold the state, which is interesting for *ConcreteObserver* objects - when they change their status, notify observers (*Observer* interface) by the update method. The *ConcreteObserver* object has a reference to the *ConcreteSubject* object and holds the state, which must be consistent with the state of the observed object (*ConcreteSubject* object) and implements the update method, which allows you to keep this consistency (Figure 3.46).

Result (Figure 3.47):

– An abstract relationship between objects such as the *ConcreteSubject* and the *ConcreteObserver,*
– Support for sending messages - the *ConcreteSubject* objects send the notification without knowing the recipient,
– Unexpected upgrade - because the fact that observers do not know about existence of other observers.

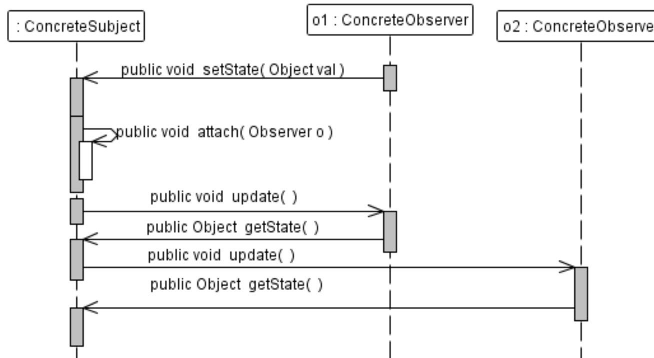


Figure 3.47. Sequence Diagram of the *Observer* Pattern.

3.4.8. State

Problem: You can change behaviour of an object when you change the inner state of this object by forming its derivative object.

Solution: The *Context* object defines an interface for clients and maintains the *ConcreteState* object. The *State* interface declares the interface associated with the state of the *Context* object. The *ConcreteState* objects represent the complete behaviour associated with the state of the *Context* object (Figure 3.48).

Client: The object, which configures the *ConcreteState* objects by using the *Context* object (Figure 3.49).

Result:

– The location of all functionality in one object implementing the *State* interface and diversification of the behaviour that depends of the change of the state in the derived classes. These classes acquire all the functionality,

– Each derivative object implements a new full functionality, which improves the visibility of transitions between states,

– Sharing objects such *ConcreteState* objects, because their states are represented by their types.

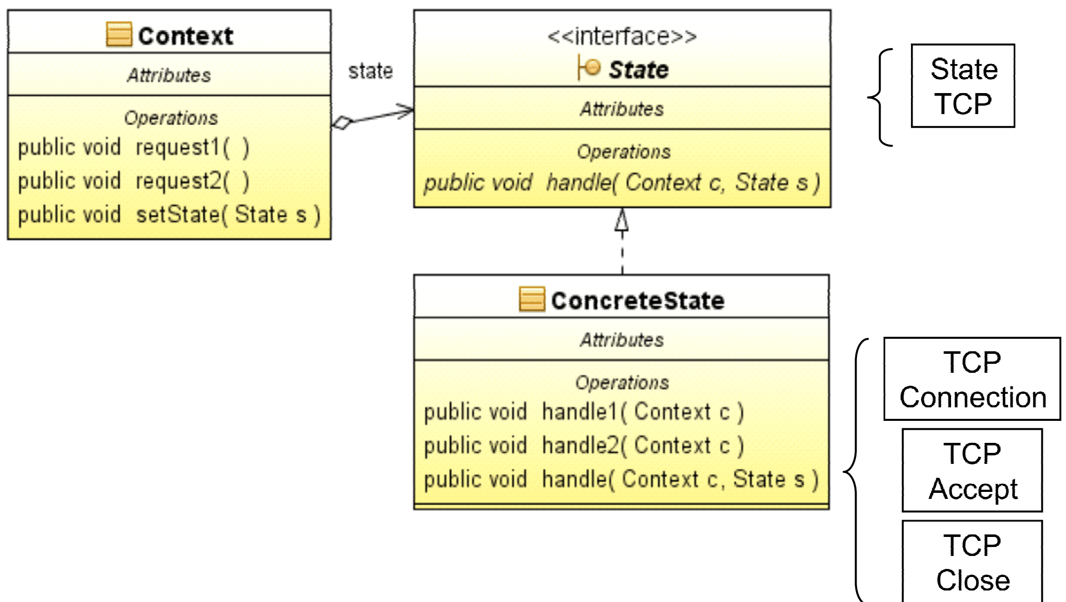


Figure 3.48. Class Diagram of the *State* Pattern.

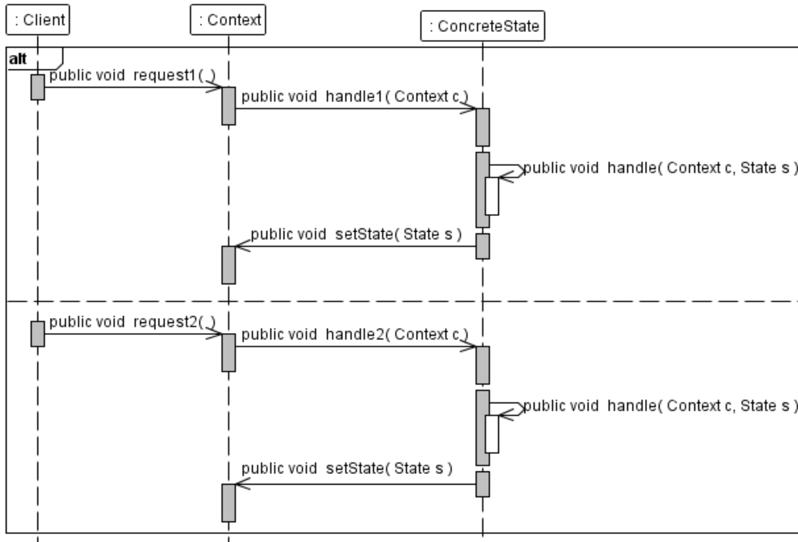


Figure 3.49. Sequence Diagram of the *State* Pattern.

3.4.9. Strategy

Problem: A selection of different business rules or algorithms for different versions depending on the context.

Solution: It selects a version of the algorithm, which separates from its implementation (Figure 3.50).

Client: The decision to implement strategies and a context of the objects takes by the owner of these objects, which provides information how to create the proper objects of the strategy and the context of using the factory object (Figure 3.51).

Result:

- Define a family of algorithms,
- Defining a *Strategy* interface which contains a method providing algorithms and methods in a *Context* class which uses these algorithms,
- Elimination of manual selections or conditional choices of an algorithm of the strategy by introducing a mechanism of polymorphism - especially, if the choice of the algorithm is not the transitional one.

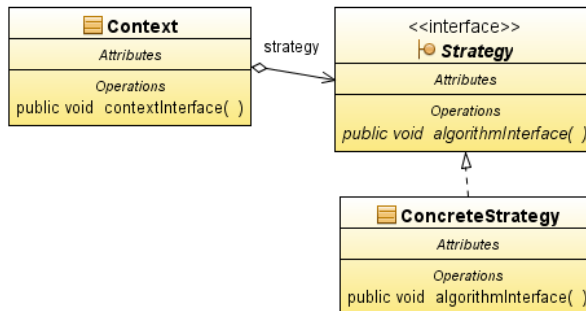


Figure 3.50. Class Diagram of the *Strategy* Pattern.

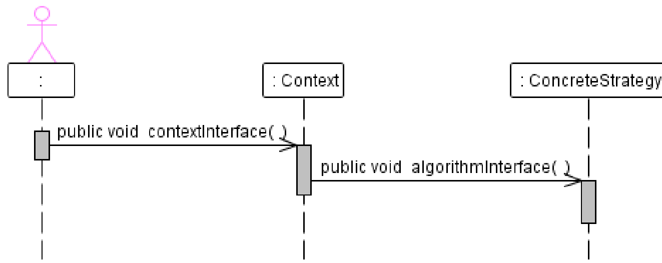


Figure 3.51. Sequence Diagram of the *Strategy* Pattern.

3.4.10. Template Method

Problem: Define a skeleton of an algorithm, providing a definition of details of this algorithm to the derived classes.

Solution: The *AbstractClass* class defines the abstract algorithm, but the same parts of the abstract algorithm are supplemented by various definitions, implemented by *ConcreteClass* classes (Figure 3.52).

Result: "Hollywood principle" (do not call us, we'll call you), where the base class method calls the methods from derived classes (Figure 3.52).

Implementation: the creation of libraries, which gives rise to common behaviour in library classes.

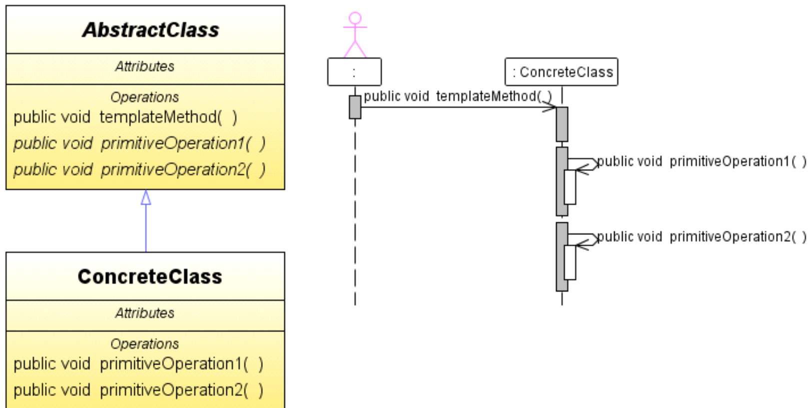


Figure 3.52. Class and Sequence Diagrams of the *Template Method* Pattern.

3.4.11. Visitor

Problem: The pattern allows you to define a new operation without changing the component class in which it operates.

Solution: An *ObjectStructure* object can enter your items and allows objects that implement the *Visitor* interface to visit your elements (which may be composite). The *Visitor* interface declares the method of visit, which receives the item as a parameter you want to visit. The *ConcreteVisitor* object implements the visit method, which allows storing information about the status of individual elements. The *ConcreteElement* objects define the accept methods which take the *ConcreteVisitor* object (Figure 3.53).

Client: The *Client* must create a *ConcreteVisitor* object, and go through the whole structure of objects represented by a *ObjectStructure* object, visiting each element, using the *ConcreteVisitor* object. Each visited element calls the visit method of the *ConcreteVisitor* object, giving access to each other, and enables him to call the appropriate method for your class (Figure 3.54).

Result:

– Easily add new operations, which are dependent on complex elements,
– Bringing together related operations in a class that implements the *Visitor* interface, and separation of unrelated ones in their subclasses,
– It is difficult to add new *ConcreteElement* classes because you need to declare a new visit method in the *Visitor* interface and new implementation of visit methods in the *ConcreteVisitor* classes.

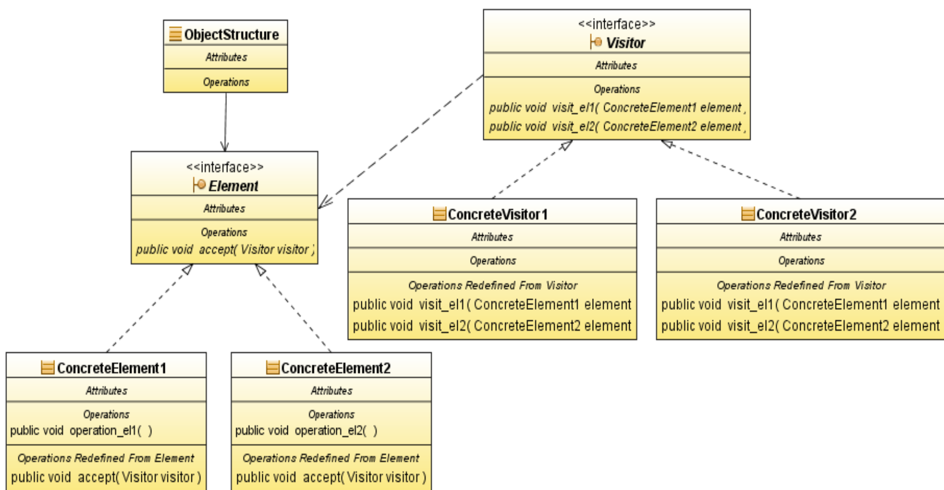


Figure 3.53. Class Diagram of the *Visitor* Pattern.

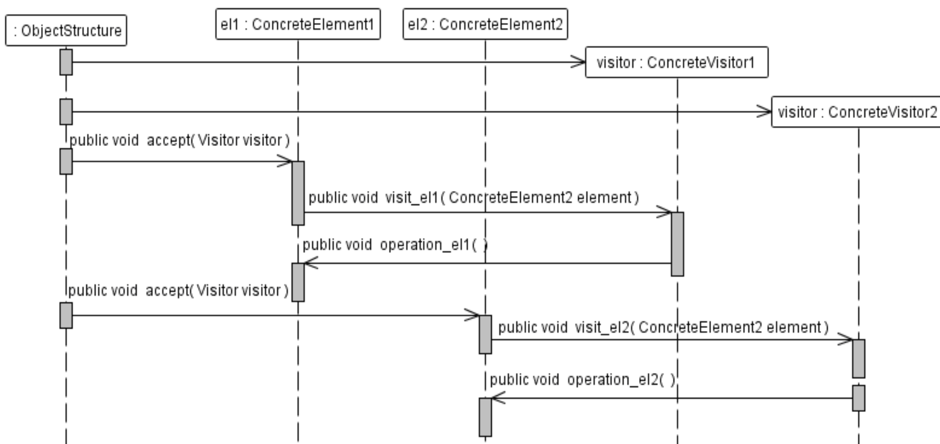


Figure 3.54. Sequence Diagram of the *Visitor* Pattern.

4. Design patterns used to build the Business Tier

Figure 4.1 shows the definition of the Business Tier of a five-tiered model of logical separation of tasks of an Enterprise application [2].

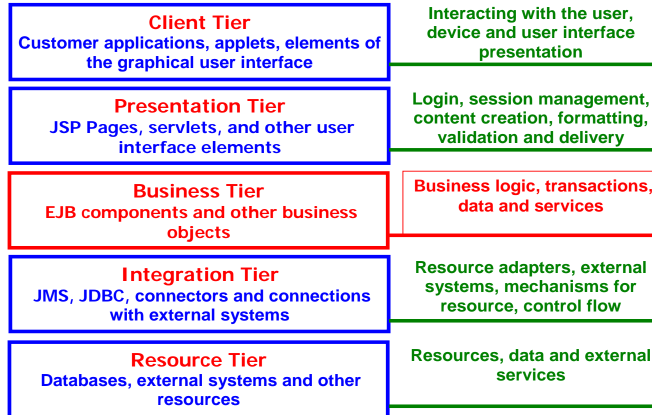


Figure 4.1. The definition of the Business Tier of multitier information system [2].

4.1. Basic issues of the Business Tier design

Basic issues of the Business Tier design are described in the subsections of Section 3.1 such as using the session components, use of *Entity* components and caching of references and handles of remote enterprise bean components [2].

4.1.1. Using the session components

The session component (EJB specification) is characterized as follows:

- It is used by one customer or user only,
- There is only for the duration of the session,
- It is destroyed during failure of a container,
- It is not a permanent object,
- The time limit of its life may be exceeded,
- It may participate in transactions,
- It can be used for modelling of communication such as stateful or stateless between the client and Business Tier components.

Stateless session components have some features as follows:

- Do not store customer data,
- Any instance of the stateless session component for any customer can be allocated from a pool of stateless session components - after servicing the customer, each component is returned to the pool,
- They are used in case of a service that requires only one method call.

Stateful session components are based on the following issues:

- Stores session state,
- Components are returned into a pool of stateful session components only when the client has completed its session,

– If the service requires a number of method calls to complete the transaction, which is a conversational process. This choice improves the scalability of the system.

There are some ways of storing session state:

– Web Applications store the session state of clients in the Presentation Tier,
– Applications, with different types of clients are keeping the session state in the Business Tier.

Managing session state is dependent on:

– Hardware,
– Network Traffic Management,
– The use of clustering for the Web container,
– The use of clusters for the EJB (Enterprise Java Bean) container,
– Linking client session to a particular server in the cluster (server affinity),
– Session Replication,
– Session Persistence.

4.1.2. Using the Entity components

Entities are distributed, shared and persistent transactional objects and EJB containers provide additional infrastructure to support the scalability, security, performance and clustering. Specification of *Entity* components (EJB specification) defines among others, following features:

– They represent an object-oriented view of life,
– They participate in transactions,
– They are shared by multiple users,
– They have a long life,
– They are resistant to failure of a container. This type of system crashes is usually invisible to the customer.

Unique keys of main components of the *Entity:*

– A complex primary key requires definition of a class that implements the Serializable interface. Attributes of this class are key attributes of the main complex key,
– The primary key class must redefine the hashCode and equals methods,
– If the primary key consists of a single attribute, then you can use the built-in type for the key.

The business logic in *Entity* components is any logic associated with the provision of services.

Checklist for business logic contained in the *Entity*:

– Will business logic introduce a new relationship between components of the *Entity*?
– Will the *Entity* component be responsible for managing the flow of user interaction?
– Will the component assume responsibility for issues that should be included in another component of the business?

The *Entity* component must contain the business logic, which is self-sufficient in terms of handling their own data and data objects belonging to the dependent objects. In this case, it has to introduce identification, extraction and transfer of business logic, which provides such dependencies to the session component.

4.2. Bad practices of the Business Tier design

There are some bad practices during development the Business Tier of application.

1. Mapping an object model directly on the model of *Entity* components:

- Impact: large container load, generation high traffic,
- Solution:
  - Refactoring,
  - Reducing the number of components - the application of the *Composite Entity* pattern,
  - Creating a tier of access to services - the use of the *Session Facade* pattern,

2. Mapping each use case at one session component:

- Effects: a large number of session components - high complexity applications,
- Solution:
  - Refactoring,
  - The *Session Facade* pattern.

3. Sharing all the attributes of components by the method of setting and collecting: set and get:

- Impact: large network traffic,
- Solution: the *Transfer Object* Pattern.

4. The client stores the data of business components:

- Impact:
  - Large network traffic,
  - Dependence on the model of client applications,
  - Difficulties in modifying the software,
- Solution: the *Transfer Object Assembler* Pattern.

5. Embedding search services on the client:

- Impact:
  - Visualization of the complexity of application for the client,
  - Redundant code - in case of changes to modify the code in many places in the application.
- Solution:
  - Refactoring,
  - The *Business Delegate* Pattern,
  - The *Service Locator* Pattern.

6. Use of components of the *Entity* as read-only objects.

7. Using the *Entity* objects as a minor component.

8. Saving an entire graph of related *Entity* components.

9. Disclosure of EJB - related exceptions to customers outside the EJB tier.

10. Stateless session component restores session state for each call.

11. Searching data using methods that return references to remote objects.

12. Mapping the relational model to the *Entity* component model.

4.3. Analysis of basic design issues

Design cases:

1. Conceal from the client program the complexity of remote communication with business service component.
2. A transparent and uniform way to search services and business components.
3. Sharing components and business services to remote clients (take control of the business objects and reduce network traffic, or improve efficiency).
4. Centralization of some business logic components and business services.
5. The object model is an implementation of the conceptual model, which is the domain model, containing relationships and business logic.
6. Use of *Entity* components to implement the conceptual domain model.
7. Transferring data between tiers of application (reducing network traffic by reducing a number of remote calls, or improve efficiency).
8. Preparing lists of objects for remote clients' applications.

4.3.1. Business Delegate Pattern

Problem 1 – Conceal from the client program the complexity of remote communication with business service components (Figure 4.2, Figure 4.3).

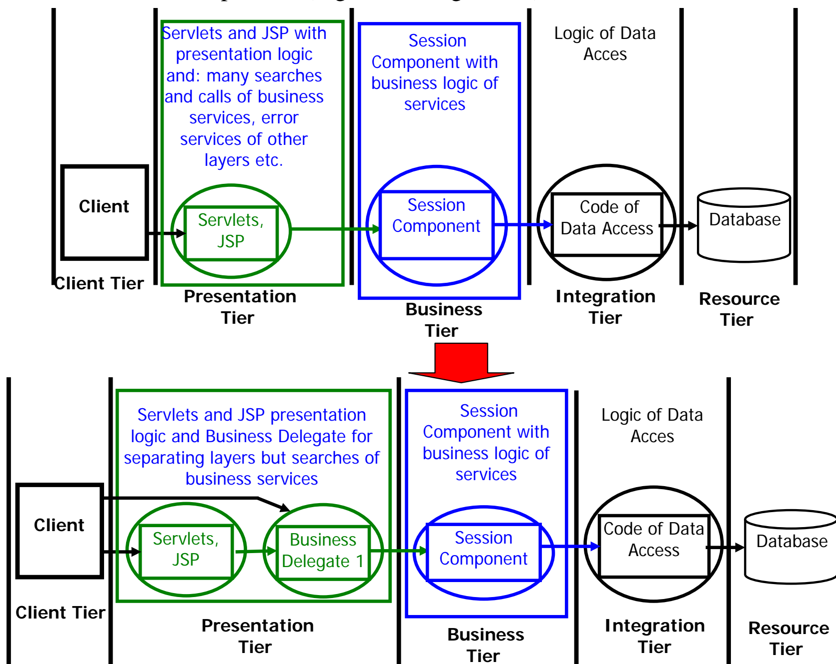


Figure 4.2. The first refactorization of the Business Tier by using the *Business Delegate* Component [2].

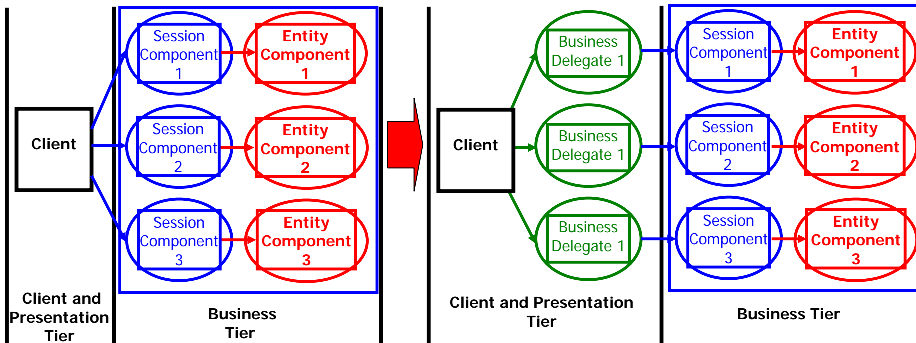


Figure 4.3. The second refactorization of the Business Tier using the *Business Delegate* Component (related to Figure 4.2) [2].

Requirements:

– Our application should have access to the Business Tier components from the presentation components and clients, which can be any device, web services and rich clients,
– Linking should be minimized relationship of clients and business services by hiding implementation details of services, for example, during searches and calls,
– Our application should avoid unnecessary calls of business services,
– Exceptions of the network and business services should change on exceptions of the presentation or client tier,
– Our application should hide in the presentation or client tier the details of the use of services, their configuration, and repeated attempts to call.

The *Business Delegate* Pattern has some characteristics:

– Abstract business service on the client side allows to hide implementation details of business services and mechanisms for their search and call,
– More changes of implementation of business services than in *BusinessDelegate* components.

The components of the pattern are as follows (Figure 4.4, Figure 4.5):

– *ServiceLocator* - implementation of the *ServiceLocator* pattern; performs searches of business services (*BusinessService* component),
– *BusinessService* - business-tier components such as the EJB component, which is used by the client. It may be part of the *SessionFacade* pattern or the JMS (Java Message Service) component.

Implementation of the pattern:

– The representative of an intermediary (*Proxy Delegate*),
– Representative adaptable (*Delegate Adapter*).

Properties:

– Reduction of dependency between the Presentation Tier and the Business Tier, hiding the implementation of the Business Tier (the client does not need to know the service name and does not need to search for services),
– Transforming the exception of business services onto exceptions easily handled in the Presentation Tier,

– Increased availability of services (temporary irregularities of the business service can occur repeatedly, outside of the Presentation Tier. Only when permanently appear system failure, the customer will be notified of the accident),
– Rendering the Business Tiers simpler for the Presentation Tier,
– caching ability of results and references to remote services, which reduces network traffic and improves application performance,
– Placing an additional tier - it can sometimes be a disadvantage.

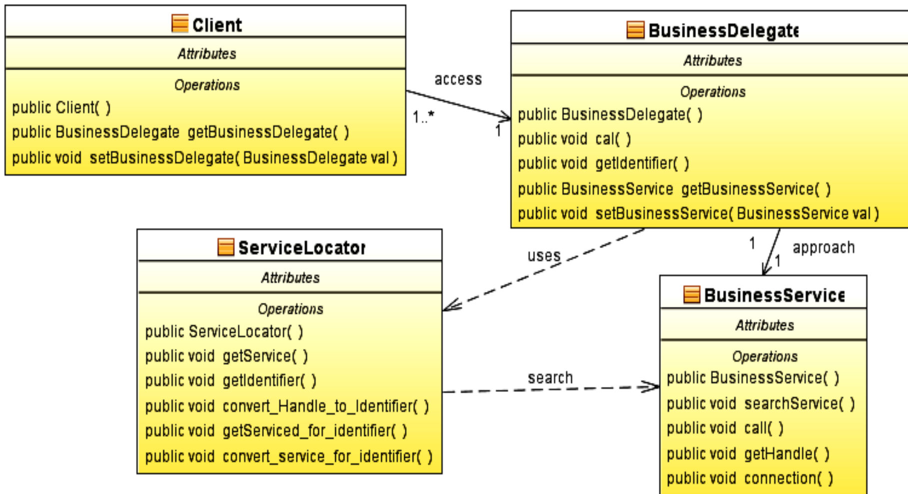


Figure 4.4. Class Diagram of the *Business Delegate* Pattern

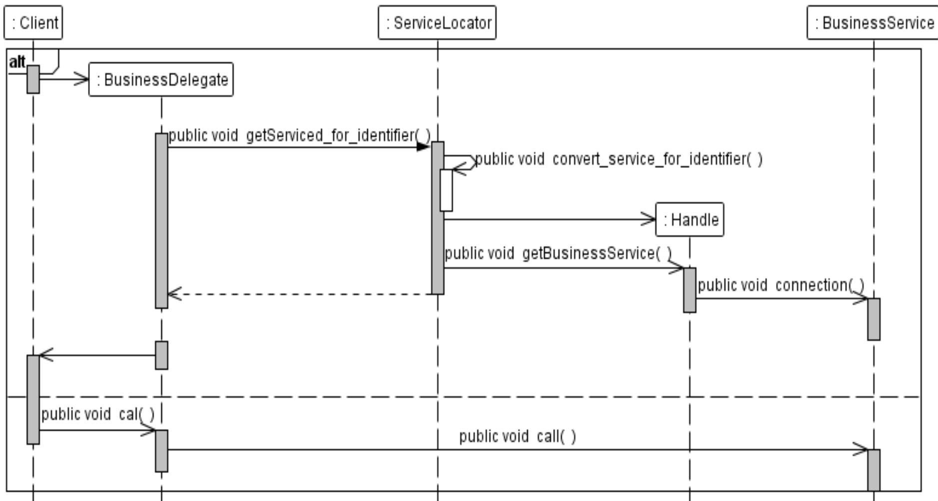


Figure 4.5. Sequence Diagram of the *Business Delegate* Pattern – a business service call.

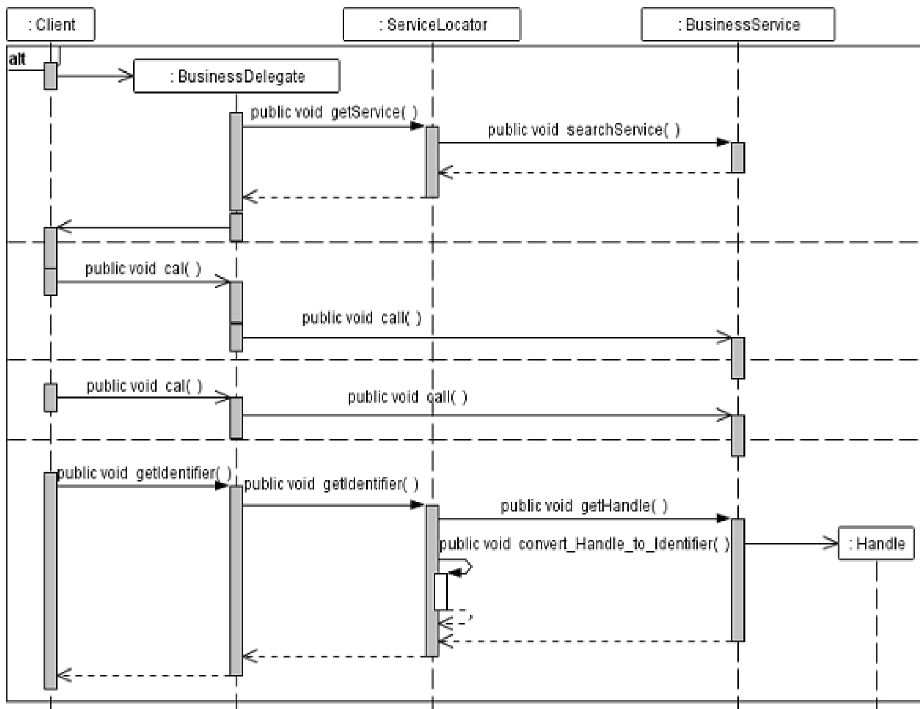


Figure 4.6. Sequence Diagram of the *Business Delegate* Pattern - a business service call and store of the business handle.

4.3.2. Service Locator Pattern

Problem 2 – Transparent and uniform way to search for services and business components (Figure 4.7).

Requirements:

- Our application should use the JNDI (Java Naming and Directory Interface) interface to search for business components (e.g. EJB or JMS) or services (e.g. data sources),
- Our application should centralize searches of business services in Client Tiers,
- Our application should hide implementation details and complexity of searched components,
- Our application should avoid the loss of efficiency associated with context creation of and search services,
- Our application should be allowed to re-use found handles of business services.

The *Service Locator*:

- In computer systems, there are two *ServiceLocator* objects - one for the Presentation Tier and the other for the Business Tier,
- Optimization of search operations and their creation.

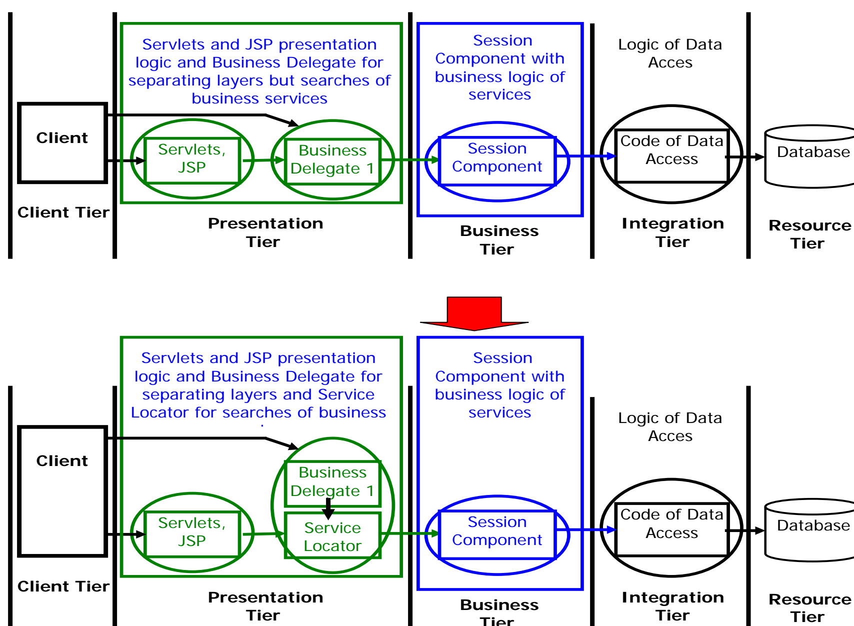


Figure 4.7. Refactorization of the Enterprise application [2].

Components of the pattern (Figure 4.8, Figure 4.9):

– *Client* - It is the *Business Delegate* component, who is awaiting access to one of the *Session Facade* components. Similarly, *DataAccess Object* components are clients when retrieving JDBC (Java Database Connectivity) data source,
– *Cache* - A cache of handles to earlier sought out services to reduce unnecessary operations, which improve performance,
– *Initial Context* - The starting point of the search process and the creation of objects. Service providers give the context object, depended on your desired service (Target object). Each supplier is specialized in the types of services (EJB, JMS),
– *Target* - The target represents a component of a service or business or the Integration Tier. They may be EJBHome components for EJB, DataSource for JDBC data source, the *ConnectionFactory* component for JMS (Java Message Service - Java EE components can create, send and read the messages in a distributed environment, controlling the long transactions),
– *RegistryService* - an object representing the implementation of the registry that stores references to services or components registered as a provider of services for objects Client).

Implementation of the pattern:

– EJB Service Locator,
– JDBC Data Source Locator,
– JMS Service Locator,
– Locator JMS queues,
– JMS Topics Locator,
– Web Service Locator,

Properties:

– Hiding the complexity,
– Providing uniform access to data,
– Facilitating the addition of EJB business components,
– Improving network performance in providing services,
– Improving performance through caching the handles of services.

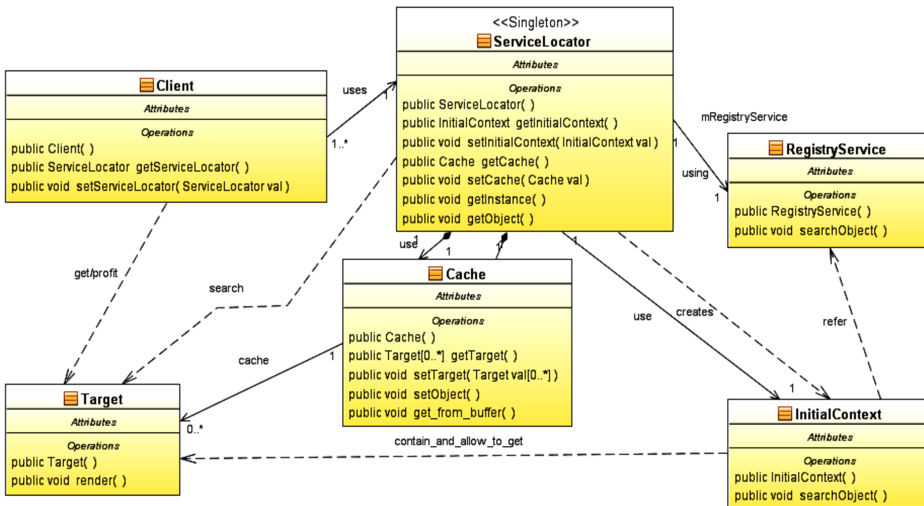


Figure 4.8. Class Diagram of the *Service Locator* Pattern.

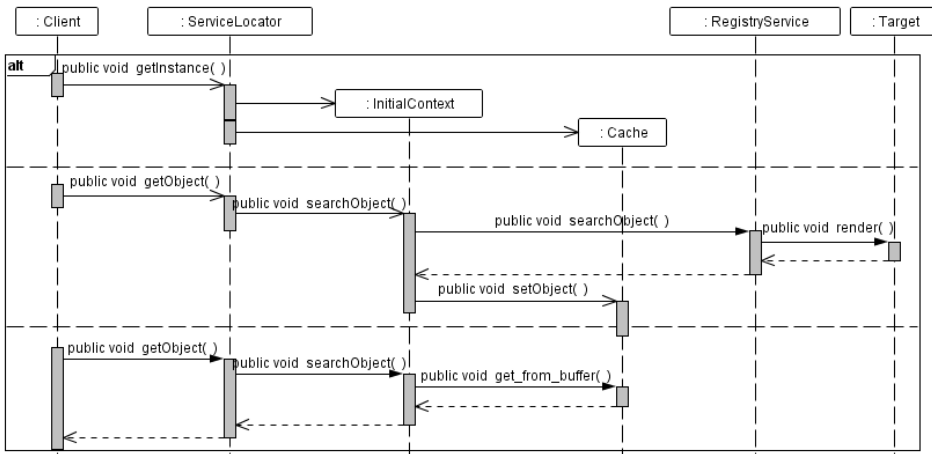


Figure 4.9. Sequence Diagram of search business services components use-case of the *Service Locator* Pattern.

4.3.3. Session Façade Pattern

Problem 3 – Provide components and business services for remote clients (take control of the business objects and reduce network traffic, or improve efficiency) (Figure 4.10).

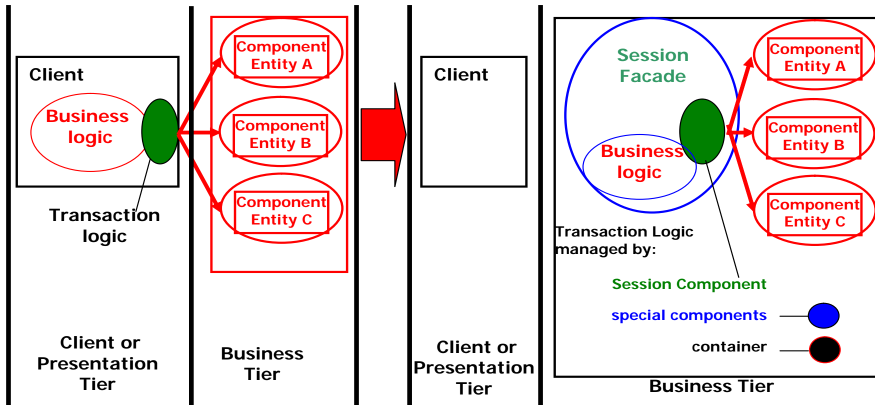


Figure 4.10. Refactorization of the Business Tier by using the *Session Façade* Pattern [2]

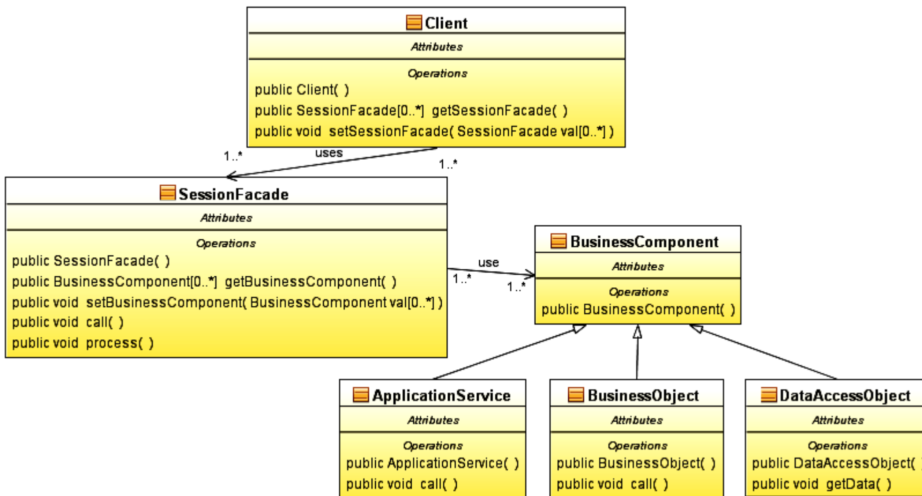


Figure 4.11. Class Diagram of the *Session Façade* Pattern.

Requirements:

- Our application should prevent clients from direct access to components of Business Tiers to counteract the establishment of too many dependencies between clients and the Business Tier (design case 1),
- Our application should provide the tier of remote access to *Business Object* components (design case 5), or other business objects,
- Application services should be grouped and provided to remote clients (i.e. *Application Service* pattern) and any other services,
- Our application should centralize and combine all of shared business logics to remote clients,
- Our application should hide complex interactions and interdependencies between the components and business services to facilitate management, centralization of logic, increase in flexibility and facilitate the change.

Components of this pattern (Figure 4.11, Figure 4.12):

- *Client* - This is usually the *Business Delegate* pattern,
- *BusinessComponent* – It participates in the execution of client requests. This may be the *Business Object* component (design case 5) as the object model of data and conduct business or as the *ApplicationService* component,
- *ApplicationService* - This component uses the business objects and implements the business logic. The *Session Facade* component may use many such objects,
- *DataAccessObject* (DAO) – This component facilitates access to a database in simple applications where there is a tier of business objects forming an object model of data.

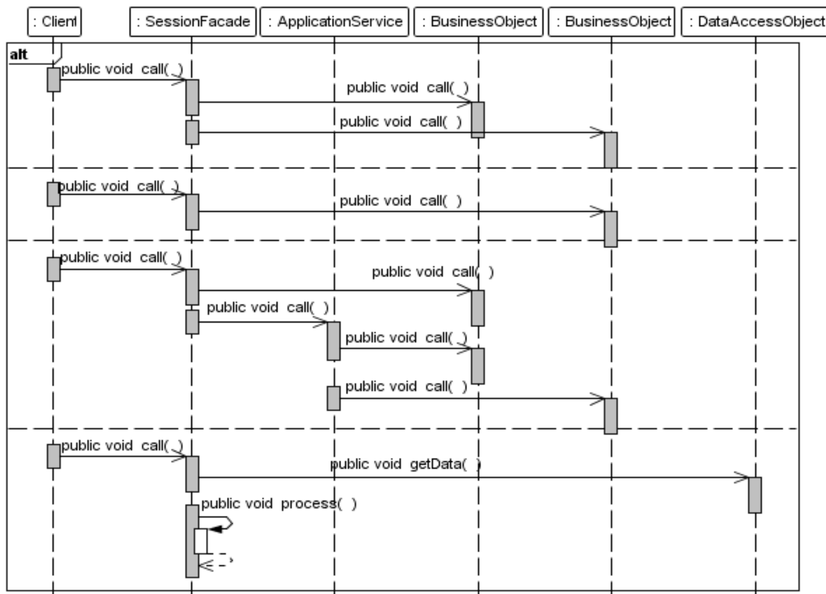


Figure 4.12. Sequence Diagram of providing business services to clients by hiding access to business components of the *Session Façade* Pattern.

Implementation of the pattern:

- Stateless session component,
- Stateful session component.

Properties:

- The introduction of a tier that provides services to remote clients,
- Providing a uniform interface,
- Reduce dependency between tiers,
- Promoting the tiered model, increasing flexibility and ease of management,
- Reduce complexity of session facade services, if applied with *Application Service* components,
- Improving productivity, reducing a number of fragmented remote methods,
- Centralization of Safety Management,
- Centralized management transactions,
- Sharing a smaller number of remote customer interfaces.

4.3.4. Application Service Pattern

Problem 4 – Centralization of some business logic components and business services (Figure 4.13).

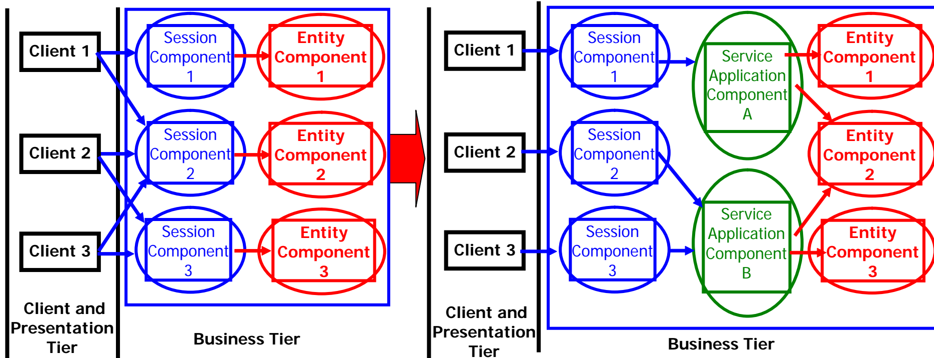


Figure 4.13. Refactorization of the Business Tier by using the *Application Service* Pattern [2].

Requirements:

- Our application should limit the amount of business logic in the facades of services,
- The business logic operates on a number of business objects or services,
- Our application should be to create a integrated services interface for the existing components and business services,
- Our application should place the logic associated with the specific use cases outside the *Business Objects* components (design case 5).

The *Application Service* – service façade:

- It provides a uniform tier of services and it is facilities of session facade services,
- It is the main business logic tier, where it does not use the *BusinessObjects* components and if necessary it uses the DAO components to retrieve business data from data sources,
- It provides a more detailed interface than the *Session Façade* components, but less detailed than *Business Object* components,
- The business logic is common to many service session façade.

The components of the pattern (Figure 4.14, Figure 4.15):

- *Client* - *Session Façade* component, the objects of ordinary Java classes (POJO), another of the *Application Service* component,
- *Application Service* – It is the leading component, encapsulates the business logic operating in several business objects, or based on a specific use case. It calls methods of *BusinessObject* or *ApplicationService* components,
- *BusinessObject* (design case 5) – These are components to complete the service of the request of the *ApplicationService* component,
- *Service* – It is a component that provides any type of service,
- *DataAccessObject* – It is a data access component without the mediation of *BusinessObject* components.

Implementation of the pattern:

– Application Service Command,
– Application Services using the GOF Strategy pattern,
– Application Service Tier.

Properties:

– Centralization of business logic used repeatedly,
– Increasing reuse of business logic,
– Prevention of duplication of code in client components,
– Simplifying implementation of session facades,
– The introduction of additional tiers within the Business Tier (centralized of common business logic).

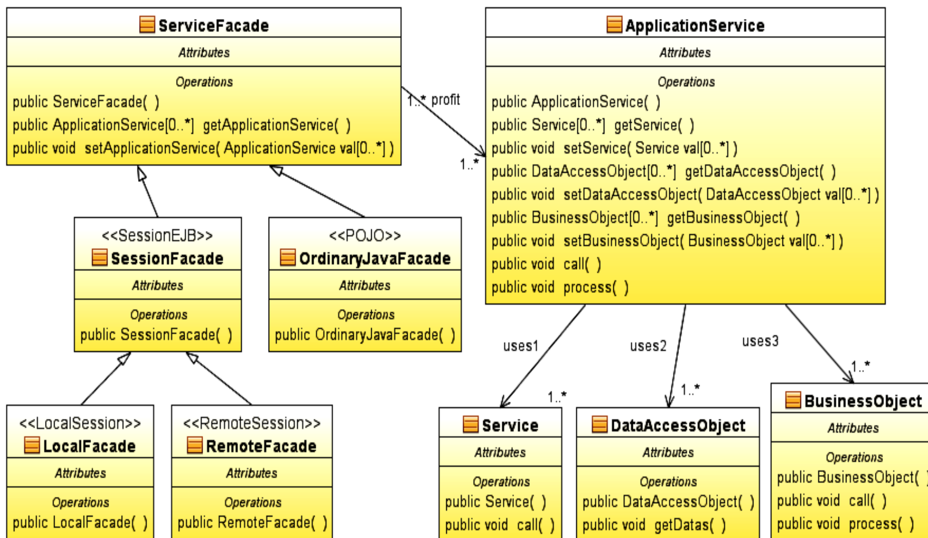


Figure 4.14. Class Diagram of the *Application Service* Pattern.

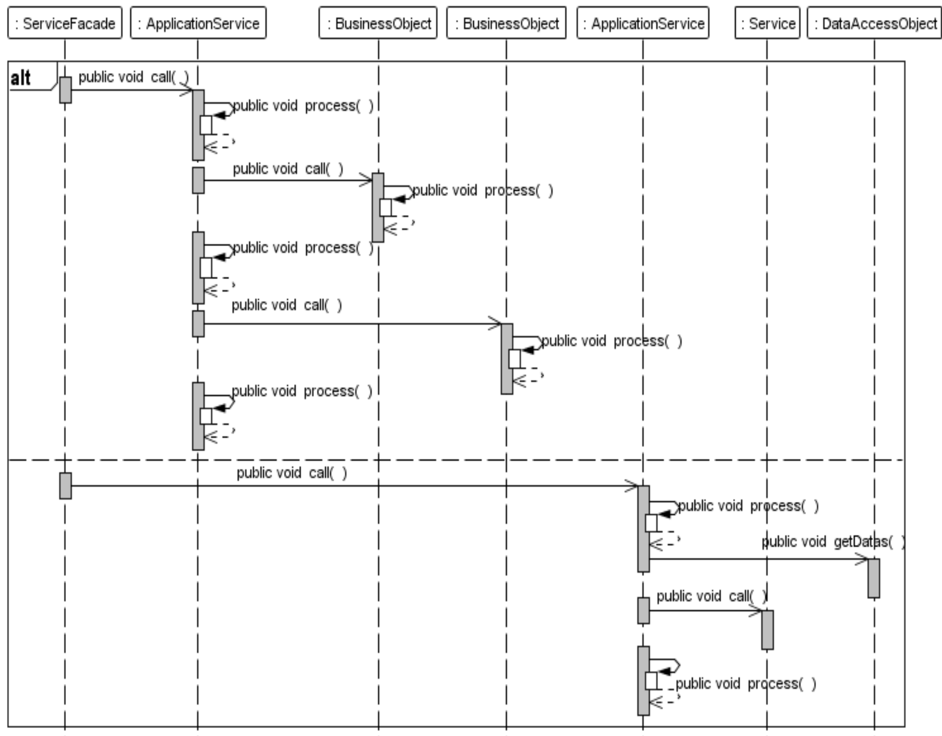


Figure 4.15. Sequence Diagram of the *Application Service* Pattern – encapsulation of the business services.

4.3.5. Business Object Pattern

Problem 5 – The object model is an implementation of the conceptual model, which is the domain model containing relationships and business logic (Figure 4.16, Figure 4.17).

There are definitions of a business model, a business use cases model, a business object model, a domain model, an object model, and a data model (The Unified Software Development Process [4]):

- The *business model* consists of two models: the *business use case model* describing the actors and business processes and the *business object model*, used to describe entities in the various use cases,
- The d*omain model* or otherwise the *conceptual model* is an abstract model that describes the main types of objects in the system. Domain objects represent events and "things" that exist in an environment where the system works. The domain model is treated as the business model,
- The *object model* is an implementation of an abstract model (Domain),
- The *data model* is used to describe a model of the implementation of such an ER model.

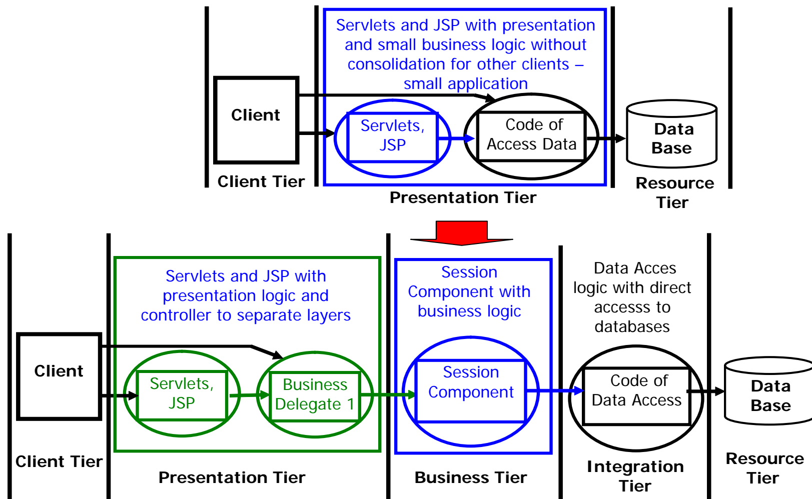


Figure 4.16. Refactorization of the Business Tier [2]

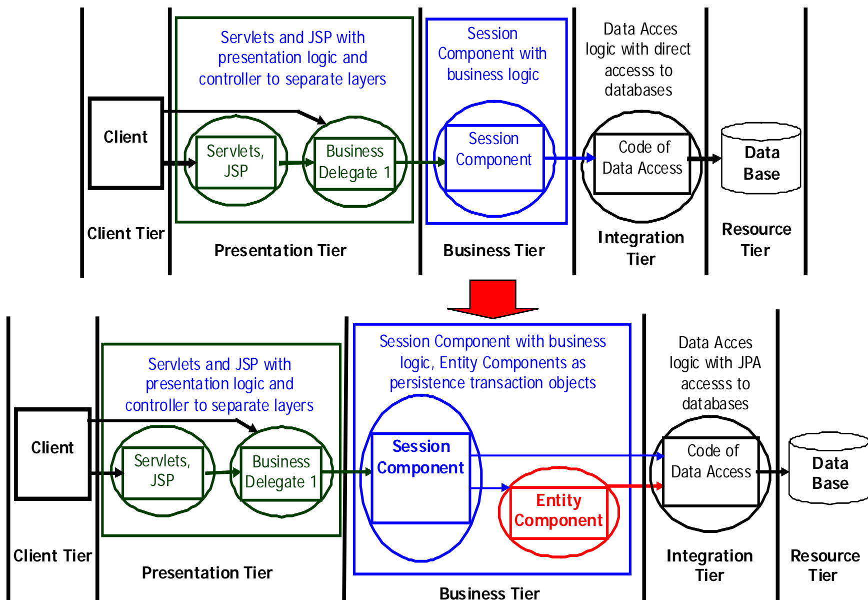


Figure 4.17. Refactorization of the Business Tier by using the *Business Object* Pattern [2].

Requirements:

– There is a conceptual model containing structural complex interrelated objects,
– There is a conceptual model with strictly defined business logic and business rules restrictions (design case 1),
– State business of the application and the related behaviour from the rest of the application should be separated, improving the consistency and ease of reuse application components,

– Our application should centralize the business logic and the state of business applications in one place,
– Our application should enhance their ability to reuse business logic and avoid code duplication.

The *Business Object*:

– It separates data and business logic using an object model - type "*Entity*".

Components of the pattern (Figure 4.18, Figure 4.19):

– *Client* – It is a client of *BusinessObject* components. This may be *SessionFacade*, *Application Service*, or each object (component) such as helped object (*View Helper* pattern) that requires access to a business object,
– *ParentBO* – It is a key element of the pattern and serves as a GOF facade. It is the main business object, the primary model of complex business objects. The parent includes its dependent objects, implements its own logic and rules,
– *DependentBO* – It is the business object that is managed by the parent *ParentBO* object during their life cycle - they cannot exist without a parent. Individual objects implement their own rules and business logic.

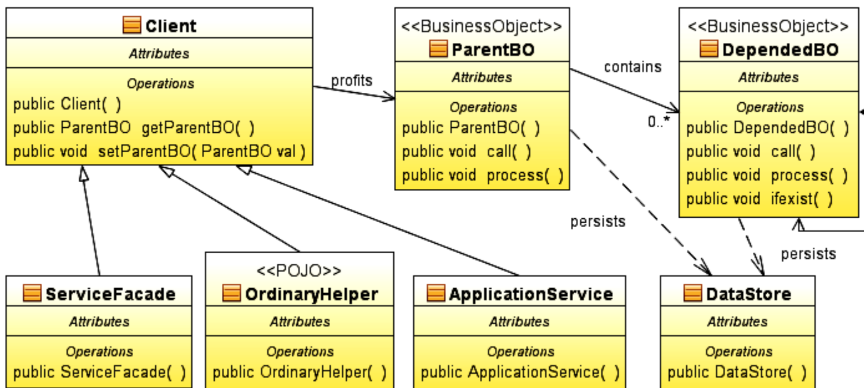


Figure 4.18. Class Diagram of the *Business Object* Pattern.

Implementation of the pattern:

– *Business Objects* as ordinary Java objects (POJO *Business Objects*),
– *Business object* in the form of the complex *Entity* component (design case 6).

Properties:

– Promote object-oriented approach in implementation of the business model,
– Centralization of business logic and state and promotion of reusable components,
– Avoiding duplication of code and turning it easier to care,
– The separation of persistence logic from business logic,
– Promoting service-oriented architecture,
– Usage of ordinary Java objects can lead to outdated information, if they do not implement their own mechanisms for synchronization and data integrity,

– Adding an additional tier - it is not necessary in cases of simple logic, or it directly benefits from the business model realized as a relational database schema. However, this may be the result of an error,
– Danger of creating very complex objects.

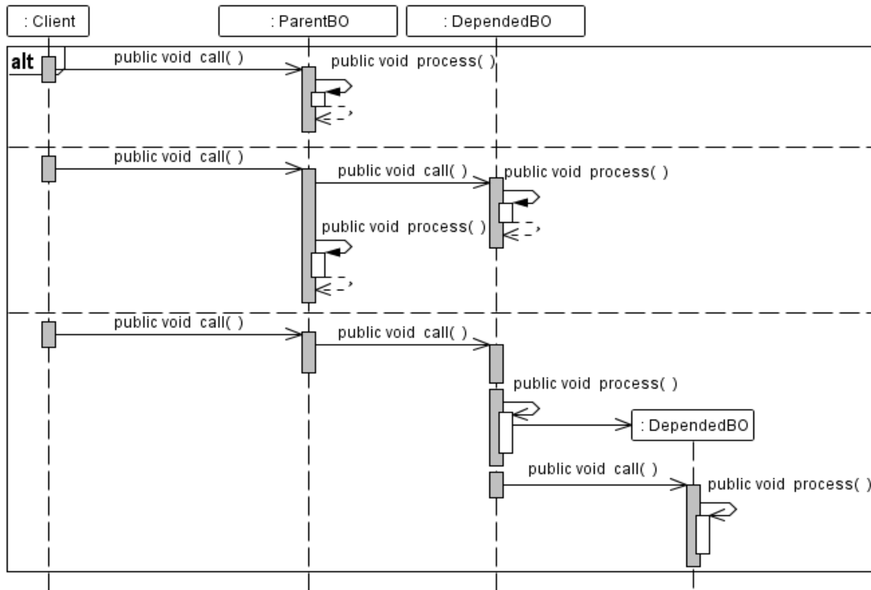


Figure 4.19. Sequence Diagram of services of the *Business Object* Pattern.

4.3.6. Composite Entity Pattern

Problem 6 – Use of *Entity* components to implement the conceptual domain model.

Business objects are not appropriate for transferring between tiers by using the *Transfer Object* component because of their remote behaviour.

A choice should be made between local and remote *Entity* components. Local components are more efficient than remote but less efficient than ordinary business objects, which implemented the *Business Object* pattern.

Requirements:

– Avoid drawbacks of remote *Entity* components, such as substantial network traffic and remote relationships between components,
– Should be used by the component-managed persistence (BMP - Bean-Managed Persistence), using in-house or non-standard implementations of the persistence mechanisms,
– Our application is in an optimal way to implement a parent-child relationships, using business objects, implemented in the form of *Entity* components,
– Our application should use and combine existing business objects implemented as POJO objects with *Entity* components,
– Our application should use the EJB container, which provides management mechanism of the transaction and security,
– Hide the physical database design from clients' applications.

The *Composite Entity*:

- It should join (using aggregation) persistent business objects in the form of local *Entity* components and ordinary Java objects (POJO),
- There are two ways of implementing the object model, related to matters of security, transaction management, resource pools, caching, and concurrency,
- Using ordinary objects of Java classes (POJO) and mechanism for meeting the specific requirements of life such as: DAO, proper implementation of the persistence mechanism using the *Domain Store* pattern or consistent with the JDO (Java Data Object) implementation,
- Application *Entity* objects according to the *Composite Entity* pattern, deciding whether to use the BMP or the CMP (Container-Managed Persistence) persistence,
- Simple applications running on the same computer can provide for clients the business objects, in the case of composite applications they can use patterns with remote calls: *Session Facade, Application Service, Transfer Data Object.*

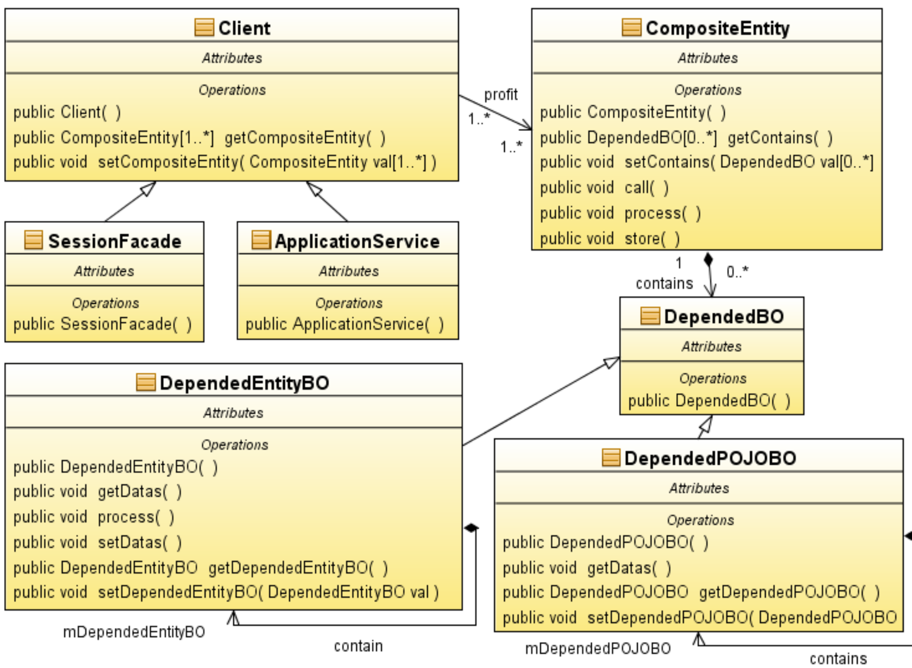


Figure 4.20. Class Diagram of the *Composite Entity* Pattern.

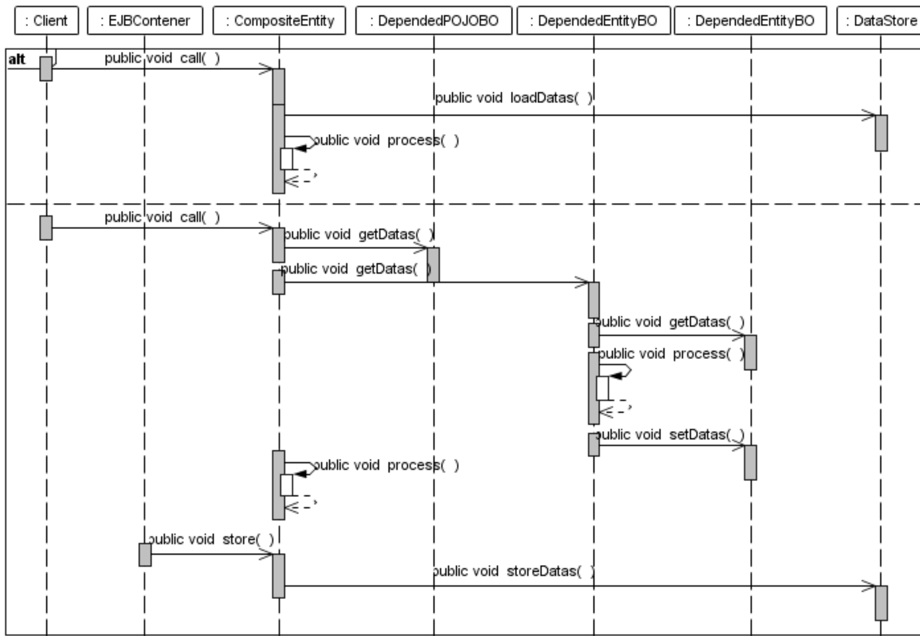


Figure 4.21. Sequence Diagram of services of the *Composite Entity* Pattern.

The components of the pattern (Figure 4.20, Figure 4.21):

– *Client* – It may be the *SessionFacade, Application Service* or other supporting components (*View Helper* component) that requires access to a business object,
– *CompositeEntity* - It is a key element of the pattern. It contains dependent objects. It plays role such as the GOF façade,
– *DependentBO, DependentEntityBO, DependentPOJOBO* – They create an object tree
– *DataStore* – It represents the persistent storage,
– *EJBContainer* – It is involved in the operations of reading and writing *Entity* components. To this end, it calls the read /write *Entity* components.

Implementation of the pattern:

– *Remote Facade* as the *Entity Composite* component in cases of simple business logic
– *Entity Composite* component with the BMP persistence,
– Lazy loading,
– Optimalization Storage,
– Complex *Transfer Object*.

Properties:

– Facilitate code maintenance,
– Improve network performance, because not all objects of the model must communicate remotely,
– Slower than the solution using standard Java objects (POJO),
– Reduce dependency on the database schema,
– Reduce the fragmentation of objects,
– Creating complex transfer objects.

4.3.7. Transfer Object Pattern

Problem 7 – Transferring data between tiers of application (reducing network traffic by reducing the number of remote calls, or improve efficiency):

- Transferring data between tiers (business objects from the Business Tier or DAO objects from the Integration Tier) should not generate much traffic, so be sure to send lots of data using one complex transfer object,
- Independence of implementation of the Presentation Tier from the Business Tier objects and the Business Tier from the tiers of integration by using the transfer object for transfering data between the tiers of an application.

Requirements:

- Our application should allow clients to access to components from other tiers and enable them to download and modify their data,
- Our application reduces the number of remote calls,
- Our application should avoid the performance degradation caused by the large number of remote calls.

The *Transfer Object*:

- These objects are used to move multiple items of data between tiers.

The components of the pattern (Figure 4.22, Figure 4.23):

- *Client* – It uses the *Component o*bject to retrieve and send data. The *Client* component is typically located in the different tier,
- *Component* - This may be an object of a different tier than the *Client* component,
- *PresComponent* – The *Component* object is located in the Presentation Tier, e.g. the BusinessDelegate component,
- *BizComponent* – The *Component* object located in the Business Tier, for example: *BusinesObject, ApplicationService, SessionFacade* components,
- *IntComponent* – The *Component* object is located in the Integration Tier of such as the *DataAccessObject* pattern,
- *TransferObject* – It is a Java object that implements the Serializable interface. It enables data transferring between the tiers.

Implementation of the pattern:

- Transfer object with modification ability,
- Many of the transfer objects,
- The *Entity* component inherited from the *Transfer Object.*

Properties:

- Reducing network load,
- Simplifying remote objects and interfaces,
- Sending more data with fewer remote calls,
- Reduce duplication of code by inheriting the *Entity* object from the *Transfer Object* component,
- Danger of obsolete transfer objects co-exitence,
- Increased complexity due to synchronization and version control for objects with modification.

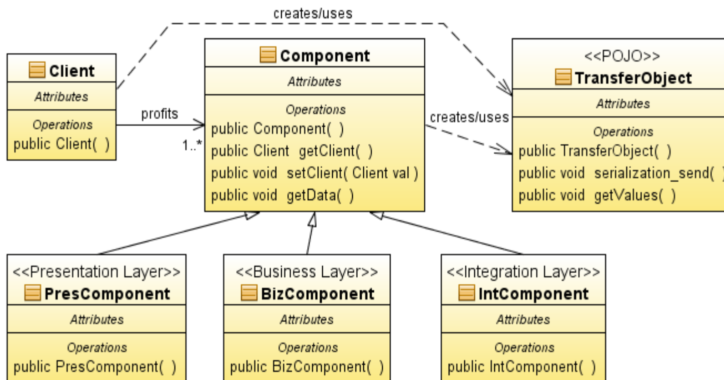


Figure 4.22. Class Diagram of the *Transfer Object* Pattern.

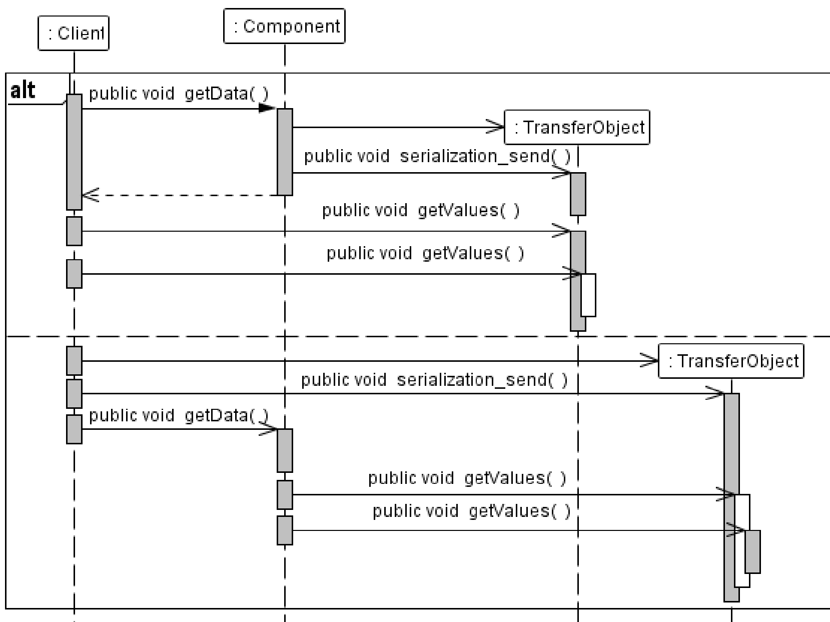


Figure 4.23. Sequence Diagram of services of the *Transfer Object* Pattern.

4.3.8. Value List Handler Pattern

Problem 8 – Making lists of objects for remote client applications:

- Enterprise applications causes many search services - an operation initiates the Presentation Tier, the Business Tier performs and the browser displays results,
- This can be done in different ways: using seek methods of the *Entity* component or by using the DAO components,
- If the response contains a lot of data, it affects the application performance degradation,
- With a large number of data, some data may remain in the Business Tier, if not all are expected by the client.

Requirements:

– Our application should avoid the overhead of using such methods as *ejbFind()* for searches that return a large number of results,
– Our application should implement a case of read-only collection of objects that do not require transaction,
– Our application should provide customers with an efficient method for searching and browsing for the long list of results,
– Results should be left on the server side.

The *Value List Handler*:

– It uses an object that implements the *Value List Handler* pattern that deals with searching, caching the results and allows customers to browse and select items from the list,
– The pattern uses the DAO component to retrieve query results from database.

Components of the pattern (Figure 4.24, Figure 4.25):

– *Client* – It is any component of the Presentation Tier or the Business Tier (component of the session), executing queries that return large result set,
– *ValueListIterator* – This object provides the iteration mechanism for browsing the contents of the *ValueList* object,
– *ValueListHandler* - It carries out search operations and gets results that are stored in the *ValueList* object,
– *DataAccessObject* – The *ValueListHandler* component uses the *DataAccessObject* component for access the data source,
– *ValueList* - It is a collection that stores the query results,
– *Value* – This object represents a single result.

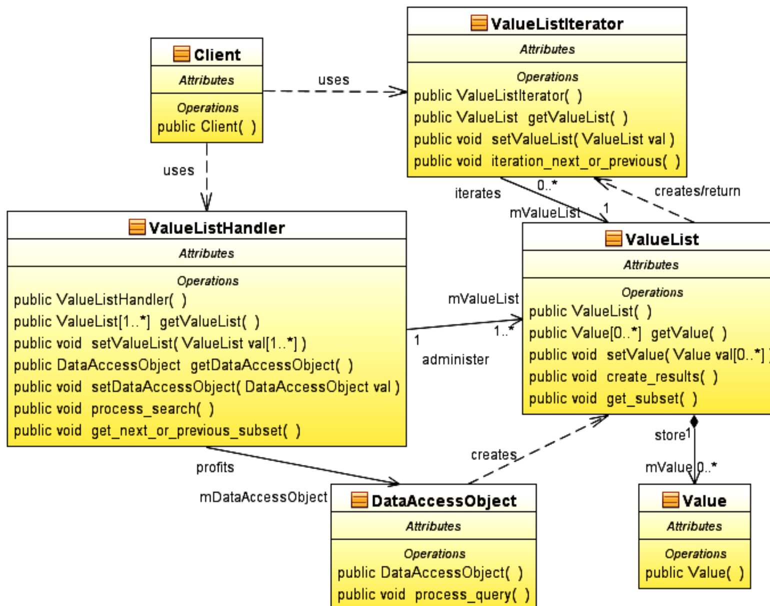


Figure 4.24. Class Diagram of the *Value List Handler* Pattern.

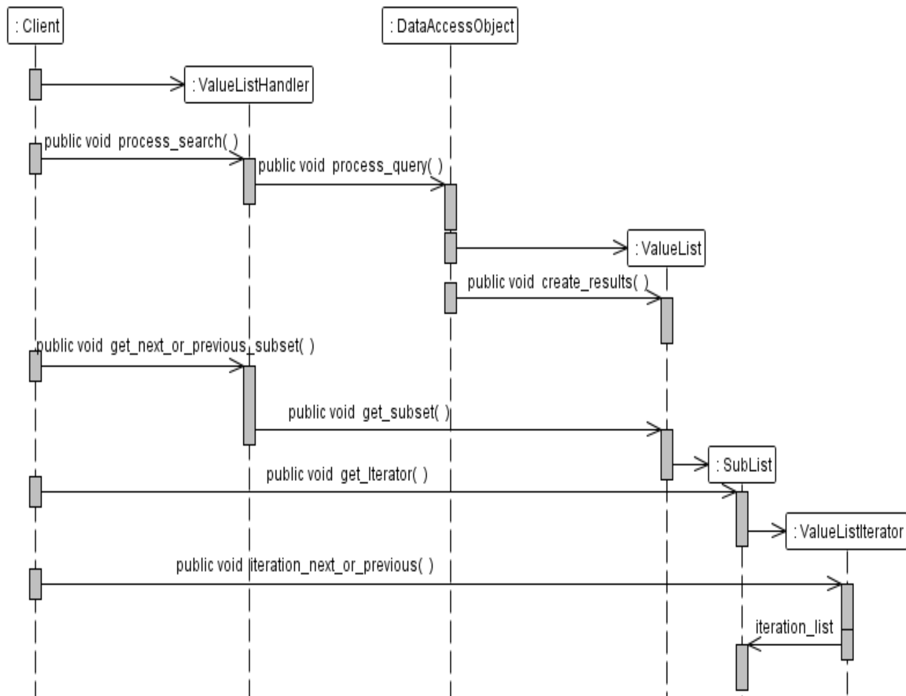


Figure 4.25. Sequence Diagram of services of the *Value List Handler* Pattern.

Implementation of the pattern:

- Using normal Java objects,
- Using *Session Facade* components,
- List of values are given from the DAO component.

Properties:

- An efficient alternative to methods such as the *ejbFind (),*
- Caching results,
- More flexible ways to search,
- Improving Network Performance,
- Avoidance of transactions for *Entity* components,
- Promoting a tiered approach and allocation of responsibility,
- Danger of creating too a large list of objects.

5. Design patterns used to build the Presentation Tier

Figure 5.1 shows the definition of the Presentation Tier of the Multitierd Information System [2].

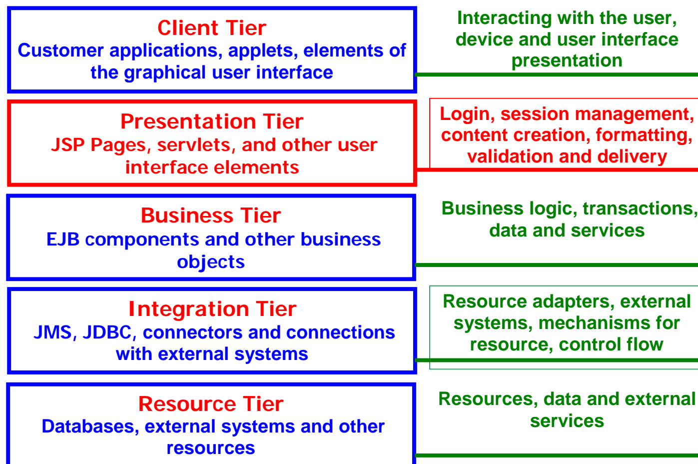


Figure 5.1. The definition of the Presentation Tier of the multitier application [2].

5.1. Basic issues of the Presentation Tier design

The main issues of the Presentation Tier are as follows:

1. Session management.
2. Control client access to applications' resources.
3. Validation.
4. Properties of helper objects.
5. Hiding resources from clients using the configuration of the container

Session management:

– Session state on the client: easy to implement, very good performance at small session data (text session data stored in the hidden form fields or cookies),
– Security of the session - a session state at the client should be encrypted if it is to be hidden from the client. This means big trouble for an Enterprise application session due to the large amounts of data,
– Session state in the Presentation Tier (the difficulty of recover the client session after server shutdown) exists when:
    - Session Time,
    - The session is cancelled,
    - State has been removed from the session.

Control client access to applications' resources - authentication and authorization:

– View protection: protection of the entire resources or their portion dependent of the different types of clients, the system state or conditions of errors,
– Protection through configuration,

– Prevent duplication of forms in order to avoid repetition of transactions - a synchronizing token stored in the user's session, direct control of the form access.

Validation:

– Validation at the client - complements the validation performed on the server should never occur alone,
– Validation on the server: bound to a form (repetition code) or based on the type of validation classes.

Properties of helper objects - the integrity and consistency (Figure 5.2):

– It should fit of the content requests from the Client Tier to the content of these objects.

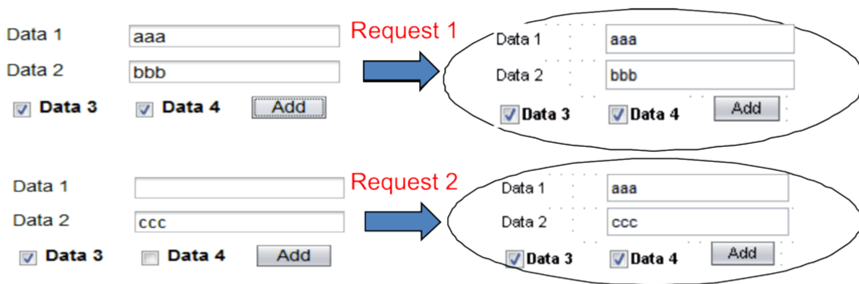


Figure 5.2. Fitting of the content requests from the Client Tier to the content of these objects [2].

Figure 5.2 shows hiding resources from clients using container configuration.

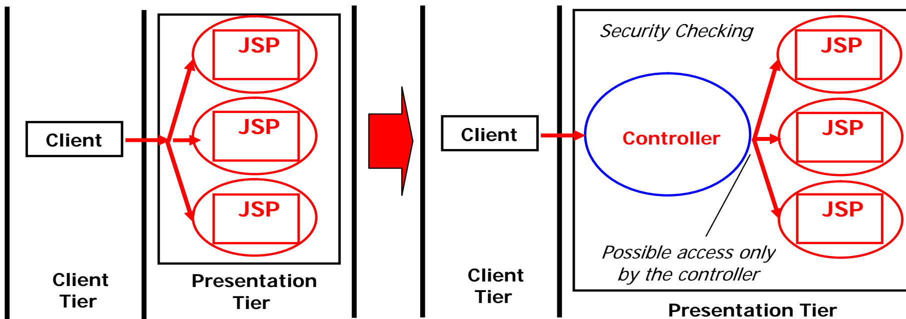


Figure 5.3. Hiding resources against the client, using the container configuration for the authentication [2].

Figure 5.4 presents hiding resources from clients, using configuration of the container and Database Users and Groups, User, Group, Role.

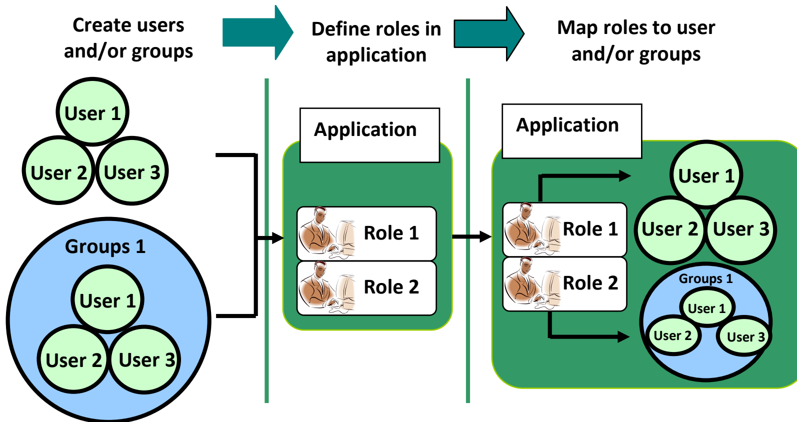


Figure 5.4. Hiding resources against the client, using the container configuration for the authentication and authorization [9].

Declarative security mechanisms - the so-called declared within "Deployment descriptors" (e.g. application descriptors web.xml for a web application).

Descriptors as external elements of applications include information specifying the security roles and access requirements mapped to roles specific to the environment, users, and security policies.

Software security mechanisms – are embedded in applications and are used to make decisions about security. They complement the declarative security mechanisms - better to express the security model of applications.

API programming mechanisms:

– *EJBContext* interface methods,
– *HttpServletRequest* interface methods. These methods allow you to make business decisions based on security roles of sender or remote receiver,
– Annotations or metadata are used to specify the information within a class code file. When an application is run, this information is used or provided by the application descriptor.

For example,

```
@ DeclareRoles ("Customer")
   public class Page1 extends AbstractPageBean {//...}
```

5.2. Bad practices of the Presentation Tier design

1. The Presentation Tier does not carry out the validation, even when validation is performed on the client tier.
2. The Presentation Tier does not carry out data conversion.
3. Data structures of Business Tiers are used by the Presentation Tier.
4. The Presentation Tier data structures are available in other tiers.
5. To transmit data between the Presentation Tier and the Business Tier or the Integration Tier, auxiliary object class are not used (called a transfer object, which is independent of these tiers).

6. It is not prohibited to send the same form twice and double triggering of the transaction, concerning the same data.
7. It does not restrict access to certain forms or parts thereof by authentication (e.g. login) and authorization.
8. It does not encrypt sensitive data sent between the web client and the www server (lack of HTTPS protocols).
9. There is a scriptlet, written in language such as Java, inside dynamically generated HTML code - this code is made available to application clients.

5.3. Design cases

1. It should intercept and modify the request and response before and after appropriate processing.
2. It should have a centralized access point for handling requests in the Presentation Tier.
3. It should avoid using specific protocol information outside its context.
4. It should centralize and achieve a modular structure of the management of actions and views of the application.
5. The view should be separated from the logic associated with its processing.
6. The view should have a modular structure, built from unit components, which combined together make up a complex page. Manage the various parts independently.
7. Major maintenance requests and business logics should be carried out before passing control to the view.
8. Views can service requests and generate responses by doing a small amount of business logics.

5.3.1. Intercepting Filter

Problem 1 – It should intercept and modify request and response objects before and after appropriate processing.

There are same issues of this problem:

- Is the client linked to the correct session?
- Has the path violated any restrictions?
- Is it supported by particular type of the web browser?
- What encryption is used by the client to send data?
- Is the data stream encrypted or compressed?

Requirements:

- Our application should centralize and process all the requests together, for example, checking the coding, storing information about each request, introducing of data compression for the message back sent to the client,
- Our application not should involve the code of the pre-processing and final processing of requests with their main code to make easier to add and remove, for example, new methods of compression,
- Independent components should be used for the processing of before and after in order to enhance their capability to re-use.

The *Intercepting Filter*:

- It is the filter of the final and pre-processing of demands actual object handling,

– It is a manager of filters, which combines basic filters connected in the chain, giving them control,
– You can create a chain of filters without changing any existing code.

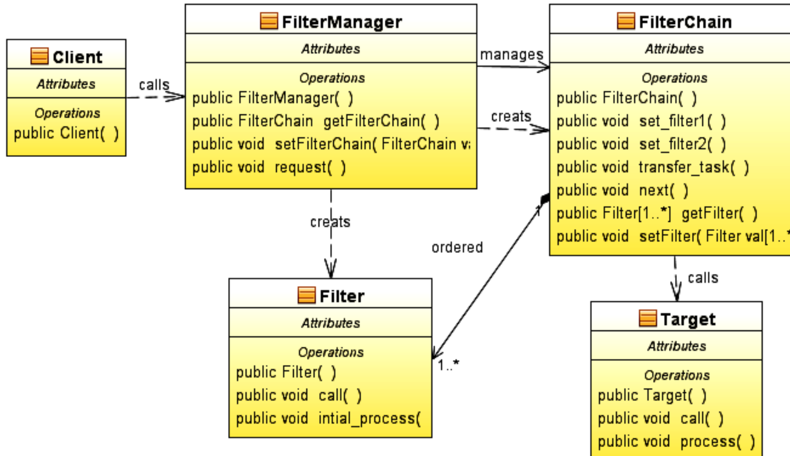


Figure 5.5. Class Diagram of the *Intercepting Filter* Pattern.

Components of the pattern (Figure 5.5, Figure 5.6):

– *Client* – sends a request to the *FilterManager* object,
– *FilterManager* – manages processing using filters. It creates an object with the appropriate chain of filters of the declared order and initiates the process,
– *FilterChain* – is the ordered collection of independent filters,
– *Filter* – represents an independent filter mapped on the path of the *Target* object. The *FilterChain* component coordinates processing of these filters,
– *Target* – is the resource dependent on a client requests.

Implementation:

– Standard filter,
– Custom filter,
– Base Filter,
– Template filters,
– Support for Web services messages,
– Custom filter SOAP,
– JAX-RPC Filter.

Properties:

– Centralization of control using independent handling procedures,
– Improved reuse,
– Declarative and Flexible Configuration,
– The small performance of data exchange between the filters.

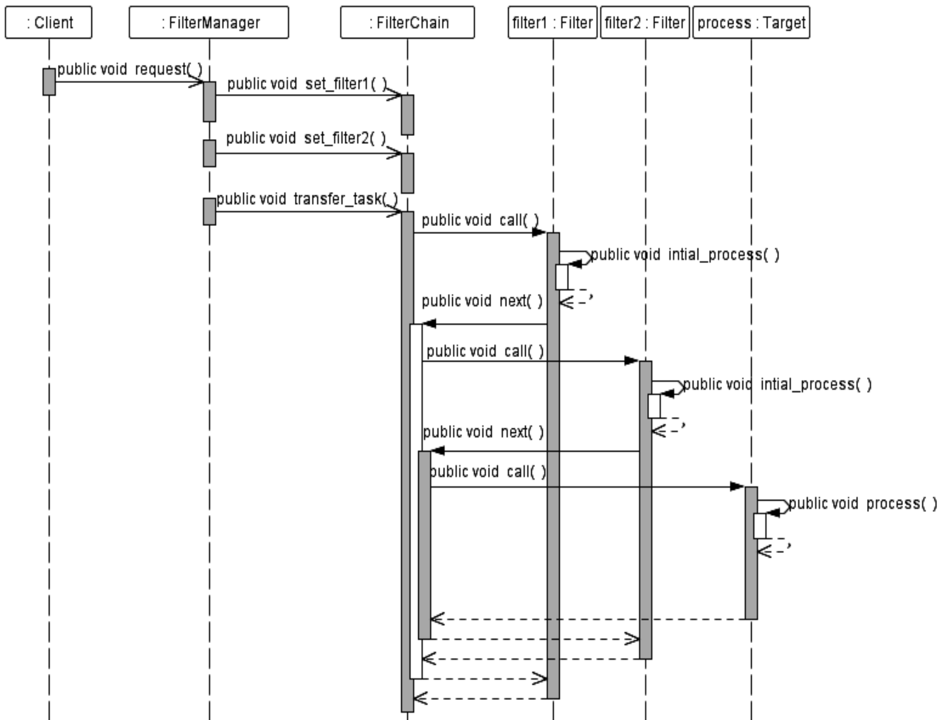


Figure 5.6. Sequence Diagram of services of the *Intercepting Filter* Pattern.

5.3.2. Front Controller

Problem 2 – It should have a centralized access point for handling requests in the Presentation Tier.

If there is no centralized access point for services, the control code is repeated in many places (e.g. views). Current solution is not modular and flexible and makes it difficult to care for code.

Definitions

- Processing Request:
  - Service of the request,
  - Protocol support and operations on the context,
  - Navigation and path selection,
  - Main processing (actions on the server),
  - Transfer of control (dispatch),
- *Processing view* - transfer of control to the components of the processing view, after the request service.

Requirements:

- Our application should avoid duplication of control logic,
- Our application should use the common logic for multiple solutions,
- Processing logic should be separated from the view,
- Our application should centralize and control all access points to the system.

The *Front Controller*:

– It is the initial point of service all requests,
– It centralizes the control logic, which in other circumstances would be certainly duplicated in different places,
– It deals with basic operations related to service requests.

The *Application Controller* is used by the *Front Controller* pattern:

– Management actions - applies to searches of the relevant services and transfer control to them in order to accommodate specific client requests (*Command* component),
– Management of views, which are returned to clients (*View* component) - applies to search and refer of the appropriate view. Although the task may be part of the *Front Controller* pattern responsibility, the use of a separate class as a part of the *Application Controller* pattern increases the modularity and the ability to reuse and makes maintenance of code easy.

The components of the pattern:

– *FrontController* – It is a place to accept requests and transfer them to the *ApplicationController* component, dealing with actions of an application and sharing of views,
– *ApplicationController* – It deals with the actions and views of the application, redirecting requests to the relevant shares and the search and selection of appropriate views,
– *Command* – It executes activities associated with handling requests,
– *View* – It represents views, which are returned to clients.

Implementation:

– Servlet receiving requests,
– JSP accepting requests,
– Command and Controller,
– Physical mapping of resources,
– Logical mapping of resources,
– Repeated mapping of resources,
– Selecting a view in the controller,
– A base class for accepting the request,
– The controller using filters.

Properties:

– Centralization of control,
– Improving management,
– Improving opportunities for reuse,
– Facilitates the separation of roles among developers.

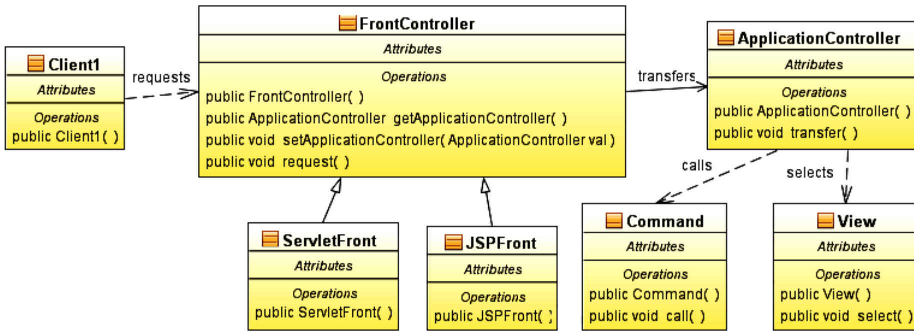


Figure 5.7. Class Diagram of the *Front Controller* Pattern.

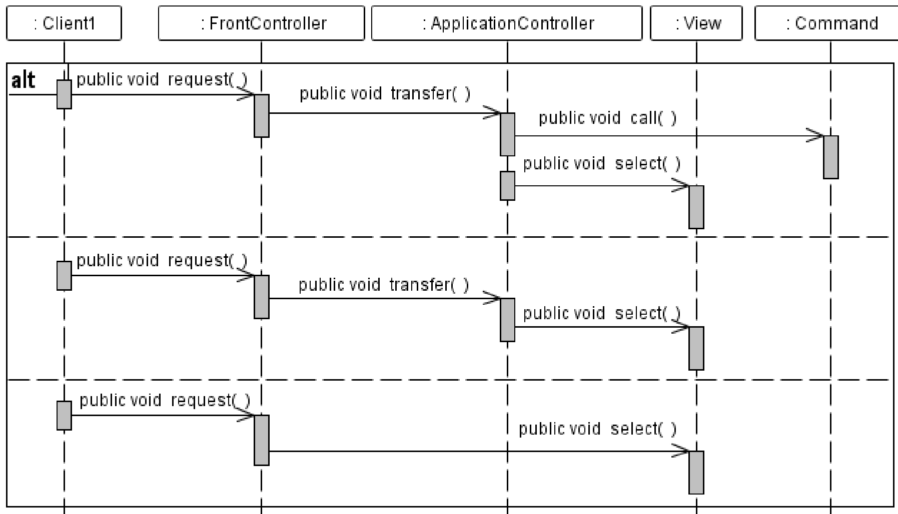


Figure 5.8. Sequence Diagram of services of the *Front Controller* Pattern.

5.3.3. Context object

Problem 3 – It should avoid using specific protocol information outside its context:

– The application uses a certain information system, such as client's request, the configuration data and data related to safety during the life cycle of request and response objects. These data are taken from the appropriate context,

– If the information does not belong to the context of the component, it should not be made available to him, because this component becomes less useful and flexible.

Requirements:

– Components and services require access to information about systems,

– The application components and services should be separated from details of the protocol,

– In the present context, our application should disclose only the necessary elements of the interface.

The *Context Object*:

– It allows the encapsulation of an application state in the independent of the protocol way,
– It transmits the state to the next elements of the application.

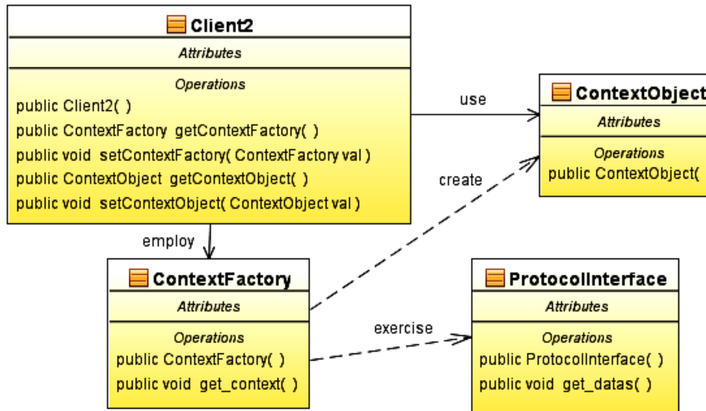


Figure 5.9. Class Diagram of the *Context Object* Pattern.

Components of the pattern:

– *Client* – It creates the *ContextObject* object by using the *ContextFactory*,
– *ProtocolInterface* – contains details of a protocol or a a tier,
– *ContextFactory* – creates the *ContextObject* object independent of the protocol or the tier,
– *ContextObject* - transmits data within the entire application, which does not fall within the area of the problem.

Implementation:

– Context of a request,
– Context of a request in the form of maps,
– Context of a request in the form of ordinary objects (POJOs),
– Verifying of the context of a request,
– Configuration context,
– Configuration of JSTL,
– Security context,
– General object of a context,
– Factory of contextual objects,
– Autofill contextual objects,

Properties:

– Improvement of maintenance and possibility of reuse,
– Facilitate tests,
– Reduce constraints related to change of interfaces,
– Reducing performance.

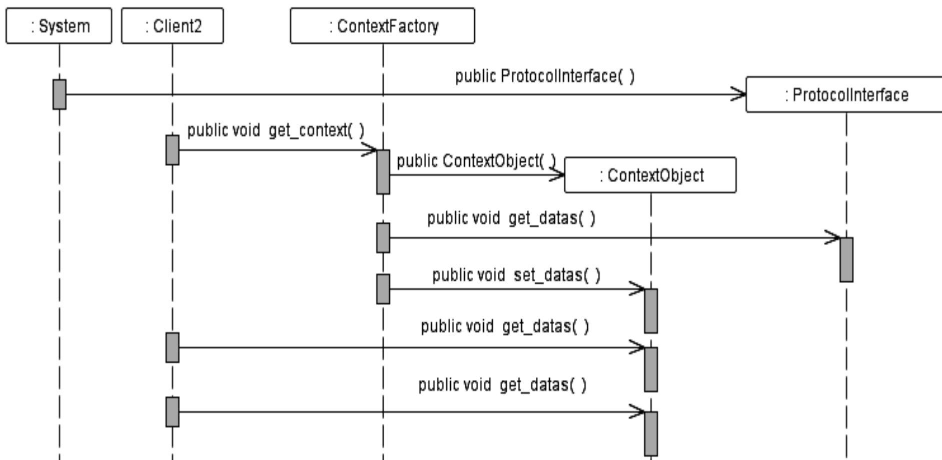


Figure 5.10. Sequence Diagram of services of the *Context Object* Pattern.

5.3.4. Application Controller

Problem 4 – It should centralize and achieve a modular structure of the action and view management of the application:

– Inside the Presentation Tier after receiving a request, it can take two decisions:
  - o The request must be forwarded for executing appropriate actions of the required service - it is management of shares,
  - o Localize and use an appropriate view - it's management of views.

– Management of actions and views can be placed in various parts of an application. Placement of this behaviour in the *Front Controller* component centralizes this functionality, but with development of applications, we must use a separate class to group the modularity increasing, extensibility and flexibility.

Requirements:

– Our application should be repeated use of code for management of actions and views,

– Expansion of request services should be improved, such as the ability to incrementally add more implementations of use cases,

– Our application should increase modularity of code to facilitate expansion of application functionality and ease testing code of individual requests regardless of environment provided by a web container or application server.

The *Application Controller*:

– Centralizes the calls of components, for example, commands and views,

– Basic aspects of service requests, for example, validation, error handling, authentication and access control can be easily attached to the so created mechanism for handling client requests.

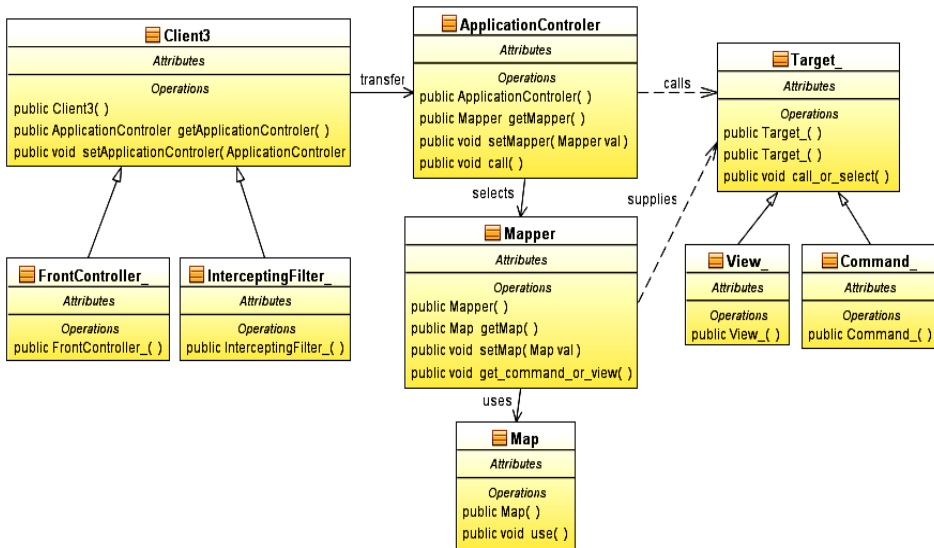


Figure 5.11. Class Diagram of the *Application Controller* Pattern.

Components of the pattern:

– *Client* – It induces the *FrontController* or *InterceptingFilter* object,
– *ApplicationController* - It *uses* the *Mapper* component to map incoming requests to appropriate actions and views, which take the control,
– *Mapper* - It uses the *Map* component to map requests to appropriate actions and views. The *Mapper* object works like a factory,
– *Map* - It keeps references to handles, which are representing target actions or views. The *Map* object can be a class or a register,
– *Target* – It is a resource associated with handling a particular request, which may be a command, view, or a style sheet,

Implementation:

– Service of the command,
– Service of the view,
– Transformation service,
– Control of flow and navigation,
– Web service:
    - a self service of SOAP messages,
    - a service of JAX-RPC messages or web services,

Properties:

Improved:

– Modularity,
– Reuse,
– Expansion.

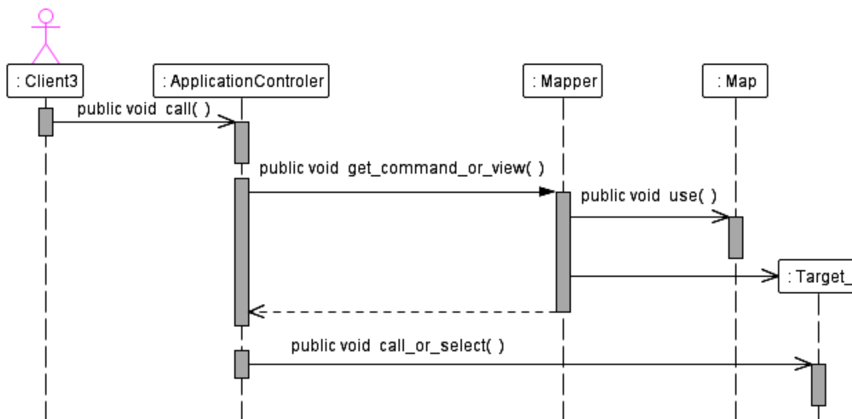


Figure 5.12. Sequence Diagram of services of the *Application Controller* Pattern.

5.3.5. View Helper

Problem 5 – The view should be separated from logic associated with its processing:

- Mixing control logic, data access and format inside a view components leads to a reduction of modularity, reuse of components, maintenance, and separation of roles of developers. Design principles are violated i.e. the separation of model from a view and control logic,
- The *Front Controller* pattern separates the logic control, but still a significant problem is separation of models and components associated with the view,
- Processing requests associated with two types of tasks: service of requests and processing of views. Processing of view can be divided into two independent stages:
  - preparation of a view: service of requests, management of actions and views (converting the request to the specific action, action called to select a suitable view, the next request will be forwarded to the desired view),
  - creating a view: a view retrieves the appropriate content of model, using the auxiliary objects to obtain data (e.g. *Transfer Object*) and their conversion to the appropriate content for the client.

Requirements:

- Our application should use templates of views, such as JSPs,
- Our application should avoid embedding application logic inside a view (avoiding the use of scriptlets),
- Application logic should be separated from a view, to clearly determine the boundaries of work of programmers and web designers.

The *View Helper*:

- Formatting code is placed in views, and process logic in the auxiliary objects,
- Views direct processing to auxiliary classes, implemented as ordinary Java objects (*JavaBean*), custom tags (*CustomTag*) files or tags (*TagFile*). Helper objects serve as intermediaries between the view and the model involved in preparation of data for display, for example, generate HTML tables.

Other goals of the pattern:

– Business logic is usually placed in the object model, for example, the *BusinessObject* or the *TransferObject* component,
– Data access code is placed in *Data Access Objects* in accordance with the standard DAO component,
– The control logic is placed in the standard *Front Controller* component and distributed to the appropriate command and supporting facilities.

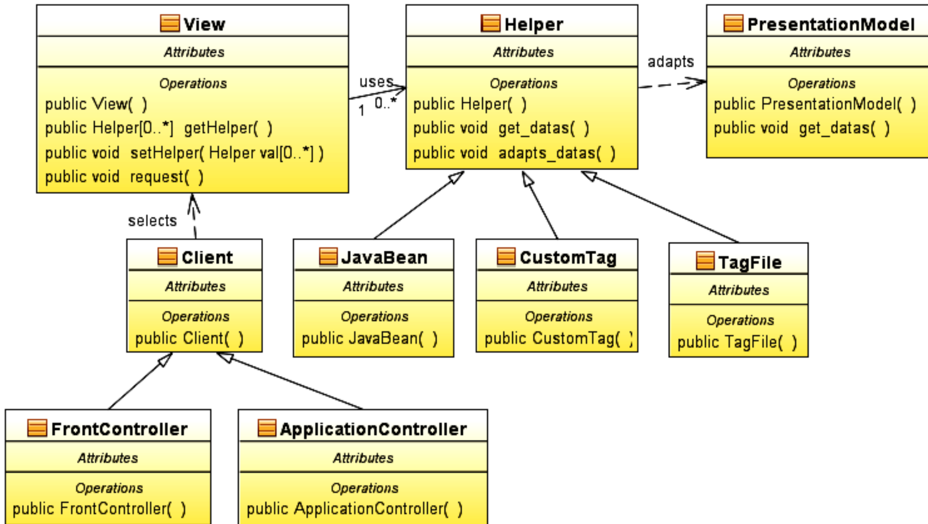


Figure 5.13. Class Diagram of the *View Helper* Pattern.

Components of the pattern:

– *FrontController* – It is used for the initial service of request (separation of control logic to the command and supporting objects),
– *ApplicationController* – It is used for simple management of views, without management of application actions,
– *View* – It includes information communicated to the client,
– *PresentationModel* - It stores data received from the business services for generating views,
– *Helper* - It encapsulates the processing logic associated with generating and formatting view. It adapts the *PresentationModel* component to the needs of display or makes data contained in the model available. Helpers are *JavaBean* components, custom tags, or markup files (JSP).

Implementation:

– Views using templates,
– View controller,
– Auxiliary *JavaBean* object,
– Custom tags,
– Assistant in the form of a Tag file,

– The auxiliary *BusinessDelegate* object.

Properties:

– Improved modularity, reuse and software maintenance facilitated (avoiding Java scriptlets, and HTML code, which is contained in code of the helper component),
– Improved separation of roles,
– Facilitate testing,
– The use of auxiliary objects are not always different from use of Java scriptlets.

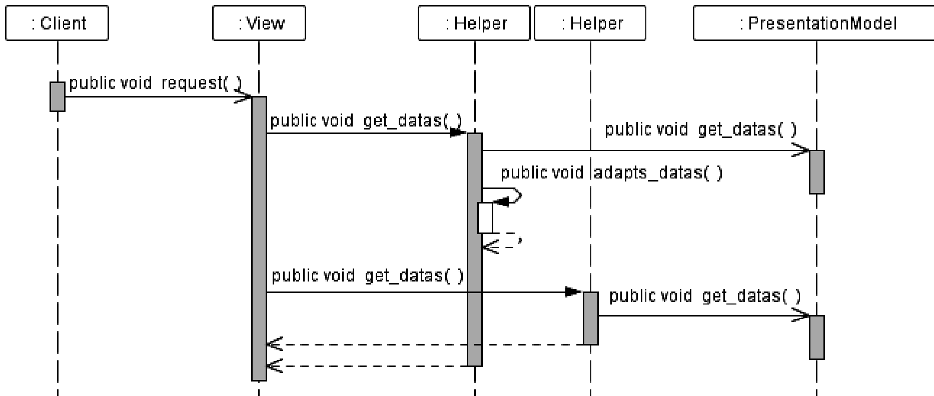


Figure 5.14. Sequence Diagram of services of the *View Helper* Pattern.

5.3.6. Composite View

Problem 6 – The view should have the modular structure, built from unit components, which combined make up a complex page. Manage the various parts independently.

Avoid duplication of code in your views, because:

– It hinders the possibility of code reuse, which reduces modularity of software - software quality is deteriorating, represented by software reuse,
– Duplicate code is harder to manage – it is deteriorating quality of the software, represented by maintainability of software.

Requirements:

– Sub-views should be common, such as headers, footers, and tables used in many views. Components can appear in different places and pages,
– The content of sub-views are subject to frequent change, or be made available only to some users, therefore it is a problem with control of access roles,
– Duplication and direct placement of sub-views in many views should be avoided, because it significantly impedes changes in the page layout.

The *Composite View*:

– It consists of many elementary sub-views,
– Each sub-view of the general template (Template) can be dynamically linked,
– The resulting layout is configured independently of the content.

Components of the pattern:

– *View* – It represents the displayed page,

– *SimpleView* – It represents a basic component of a complex view, as sub-view or segment of view,
– *CompositeView* – It consists of several *View* objects, each of which may be the *SimpleView* or the *CompositeView* component,
– *Template* - It represents a template of the view,
– *ViewManager* - Template object is used to create a page layout. Simple object uses JSP tags to include the sub-views within the template. Complex object uses the auxiliary components to manage the content and layout pages.

Implementation:

– Managing views with:
  - JavaBean components,
  - standard markers,
  - custom tags,
  - transformation,
  - Early binding of resources,
– Late binding of resources.

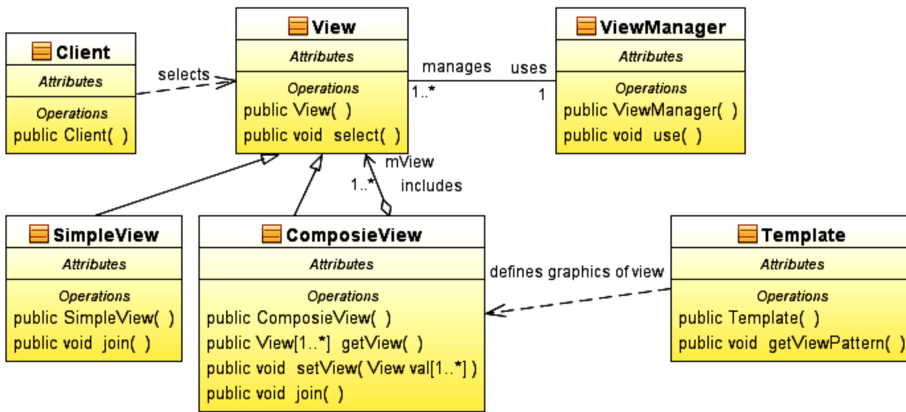


Figure 5.15. Class Diagram of the *Composite View* Pattern.

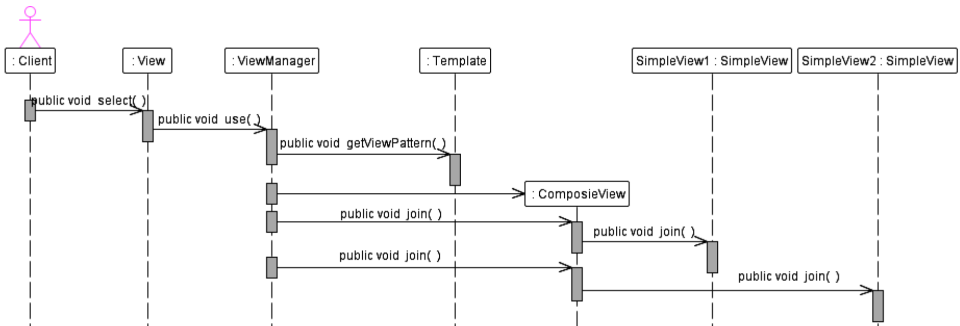


Figure 5.16. Sequence Diagram of services of the *Composite View* Pattern.

Properties:

- Improving modularity and opportunities of reuse,
- Adding a control scheme based on roles or rules,
- Facilitating maintenance of the code (based on separating a template from a view),
- Handicap code management (possibility of mismatches of sub-views),
- Performance reduction when generating sub-views.

5.3.7. Service to Worker

Problem 7: Major maintenance requests and business logic should be carried out before passing control to a view.

It should be taken into account:

- Complexity of control logic,
- Dynamics of an answer,
- Complexity of business logic and model.

Requirements:

- Perform business logic to handle requests and retrieve data that will be used to generate a dynamic response,
- Contents of the view depends on the response received after performing the business services,
- Library or a skeleton of a presentation should be used.

The *Service to Worker*:

- It is the most common method of designing the Presentation Tier, where the main role plays the *Application Controller* pattern,
- It allows centralizing control and it handles requests for download the model of the presentation before passing control to the view,
- View generates a dynamic response based on the presentation model.

Components of the pattern:

- *Front Controller* – It is used for the initial service of a request (the separation of control logic to the command and supporting objects),
- *Application Controller* – It is used for simple management of views, without management of application actions. For simple applications it can be replaced with the *FrontController* component,
- *View* – It represents a transfer of customer data, adjusted using the *View Helper.* It may be the *Composite View* component,
- *BusinessHelper* – It starts handling business demands,
- *View Helper* – It retrieves and adapts the presentation model to generate the creation of a view,
- *PresentationModel* – The presentation model stores data received after a call to business services and used when rendering a view,
- *BusinessService* – It represents the business logic. Access to remote services is done using *BusinessDelegate* objects.

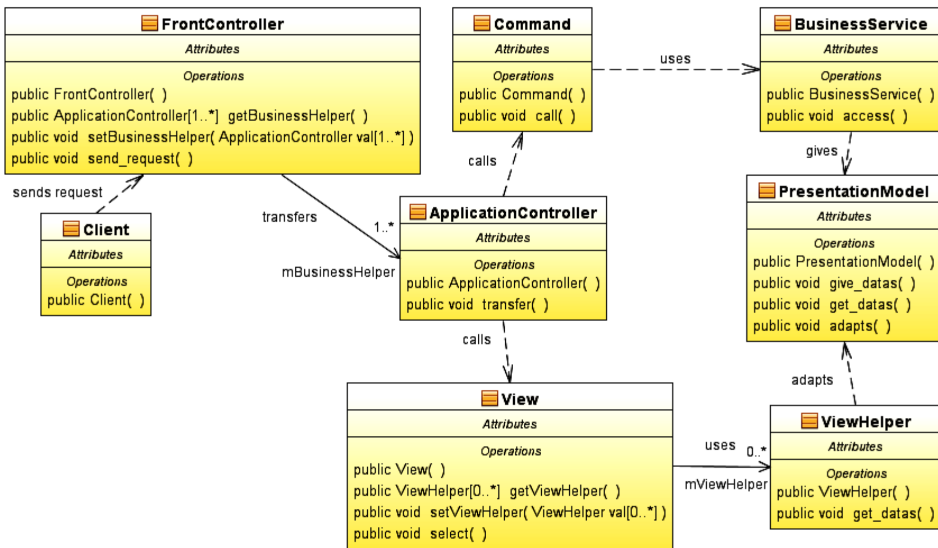


Figure 5.17. Class Diagram of the *Service to Worker* Pattern.

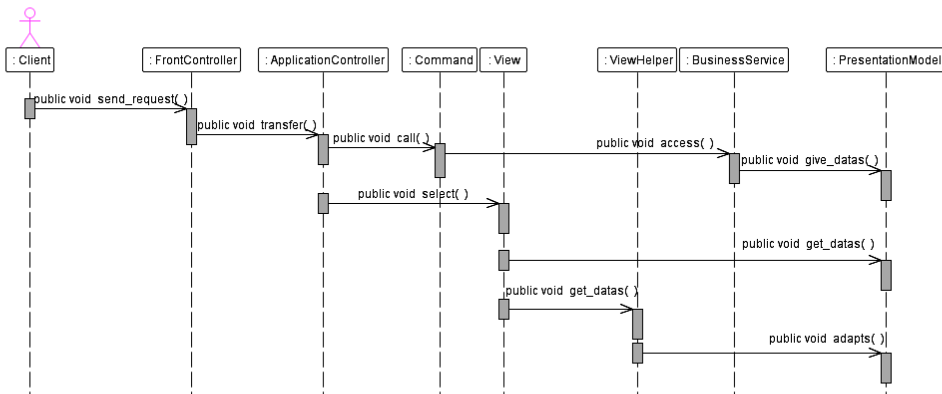


Figure 5.18. Sequence Diagram of services of the *Service to Worker* Pattern.

Implementation:

– Servlet receiving requests,
– JSP page receiving a request,
– Views using templates,
– View-based controller,
– Auxiliary JavaBean components,
– Auxiliary own tags,
– Selecting a view in the controller.

Properties:

– Centralization of control and improve modularity, reusability, and ease of maintenance,
– Proper separation between the developers roles.

5.3.8. Dispatcher View

Problem 8 – The view can perform service of a request and generates a response by doing a small amount of business logic:

- In some cases, there is few business logic to run, or not at all, before a view has been created. Usually the view is static or is a generated from an existing model of presentation in these situations,
- The view derives limited benefit from business services and access to data.

Requirements:

- Views are static,
- Views are generated from an existing presentation model,
- Views are independent of calls of business services,
- Much business processing.

The *Dispatcher View*:

- The second most common method of designing the Presentation Tier, where the main role plays the *View* component. Applications based on the *Service Worker* pattern, can use this pattern (e.g. Struts),
- Views are initial points of requests services,
- There is the small amount of required business processing performed by the view,
- Two applications:
  - the answer is completely static or a plain HTML,
  - the answer is dynamic, but it is completely generated from the existing model of a presentation.

Components of the pattern:

- *Front Controller* – It is used for the initial service of a request (the separation of control logic to the command and supporting objects),
- *Application Controller* – It is used for simple management of views, without management of application actions,
- *View* – It represents a transfer of customer data, adjusted using the *View Helper* component,
- *BusinessHelper* – It starts handling business demands,
- *View Helper* – It retrieves and adapts the presentation model to generate the creation of a view,
- *PresentationModel* – The presentation model stores data received after a call to business services and used when rendering a view,
- *BusinessService* – It represents the business logic. Access to remote services is done using *BusinessDelegate* components.

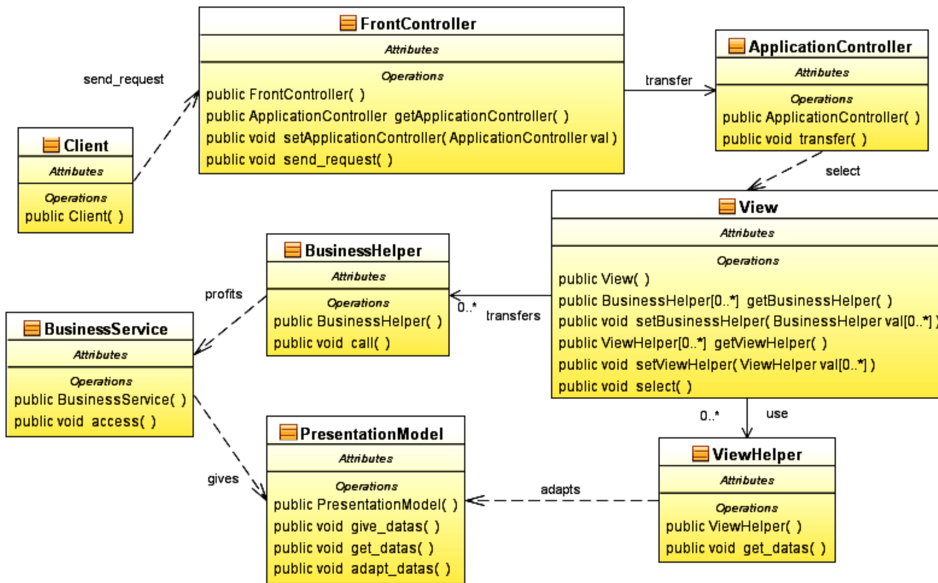


Figure 5.19. Class Diagram of the *Dispatcher View* Pattern.

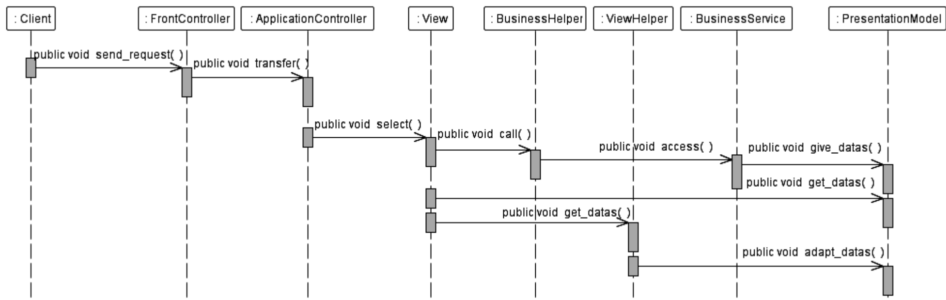


Figure 5.20. Sequence Diagram of services of the *Service to Worker* Pattern.

Implementation:

- Servlet receiving requests,
- JSP page receiving a request,
- Views using templates,
- View-based controller,
- Auxiliary JavaBean components,
- Auxiliary own tags,
- Selecting a view in the controller.

Properties:

- It is used in the presentation skeletons and the library (e.g. JSTL),
- It may result in poor separation of a view from the model and control logic,
- Allows the separation of processing logic from the view, and facilitates code reuse.

6. Design patterns used to build the Integration Tier

Figure 6.1 shows the definition of the Integration Tier of the Multitiered Information System [2].

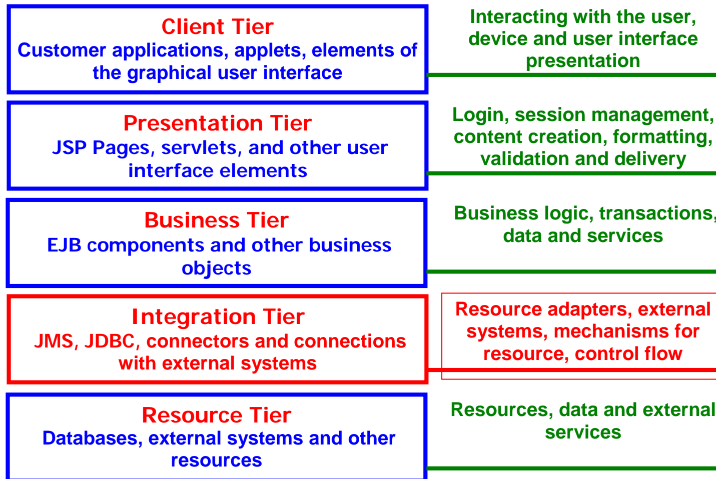


Figure 6.1. The definition of the Integration Tier of the Multitier Information System [2].

6.1. Basic issues of the Integration Tier design

The basic issues of the Integration Tier are as follows [2], [5]:

1. Managing access to data.
2. Manage connections to a database - the pool calls.
3. Managing access to data:
4. Data access code is embedded in the Presentation Tier or the Business Tier, which is used for other purposes such as:
    a. Servlet,
    b. EJB component.
5. This leads to load these classes with additional functions, impairs performance and scalability. It would be better to introduce the Integration Tier and should place the code to access data there.
6. Data access code is located in a tier of integration, but does not cache the results obtained during various access operations to the databases.
7. If you need to increase scalability and improve performance, cache processing results in a database, avoiding duplication of operations directly in databases.
8. Data access code is extracted from classes that are used to meet the other objectives.
9. Data access code should be placed logically and physically closer to its data source.

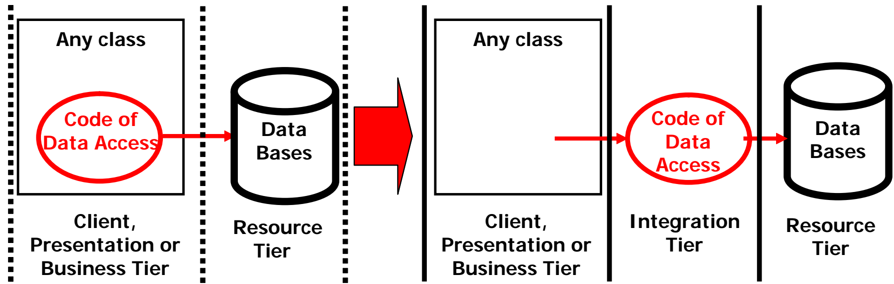


Figure 6.2. Refactorization of Client, Presentation Business Tiers based on elimination of the Integration Tier for Data access [2].

Manage connections to a database - the pool calls:

- Each client application manages a connection of databases using the same classes that are often used for other purposes, within the Presentation, Business or Integration Tiers. Database connections are not shared, which decreases performance and scalability,
- It should introduce a pool of connections initiated by clients before the connection application and put it in a tier of integration - it improves performance and scalability,
- Number of calls to data access code (DAO) within the database is limited,
- Connections to the data access code (DAO) are not always used, but are kept as open database connections and resources,
- Connection Pooling to the data access code (DAO) allows to manage reasonably connections to the database application.

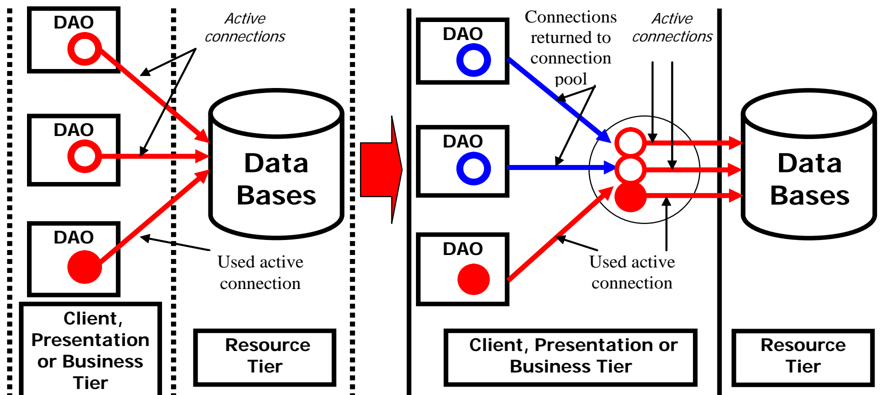


Figure 6.3. Refactorization of Client, Presentation or Business Tiers based on elimination of connections to a database - the pool calls [2].

6.2. Bad practices of the Integration Tier design

There are some bad practices during the Integration Tier design:

1. Validating data,
2. Data structures from the Presentation Tier are available,
3. Data structures from the Business Tier are available,

4. Sharing objects of an active connection to the database with Presentation and Business Tiers,
5. Direct integration of exception handling in the tiers of the presentation and business,
6. Transmission data between Presentation and Business Tiers and the tier of integration is not used by an auxiliary object class (called an object transfer, independent of the Presentation Tier and the Business Tier and data structures stored in data warehouses),
7. No encryption of sensitive data.

6.3. Analysis of basic design issues

Basic design issues of Integration Tier are as follows:

1. Hiding data access logic in a separate tier,
2. Separation of persistence mechanisms from an object model.

6.3.1. Data Access Object

Problem 1 – Hiding data access logic in a separate tier:

– Small applications do not use the *Entity* while in session facade object encapsulates business logic and the processing is carried out directly on a permanent storage,

– Most business applications use as permanent storage relational database (RDBMS), external mainframe systems, repositories, object-oriented databases (OODB), normal file systems, such as external services, B2B or services by credit card. Each system uses different access mechanisms supported by the API and has a different functionality,

– Mixing data access code with application logic leads to a relationship between an application and permanent storage, which requires many changes in the application if you change the database.

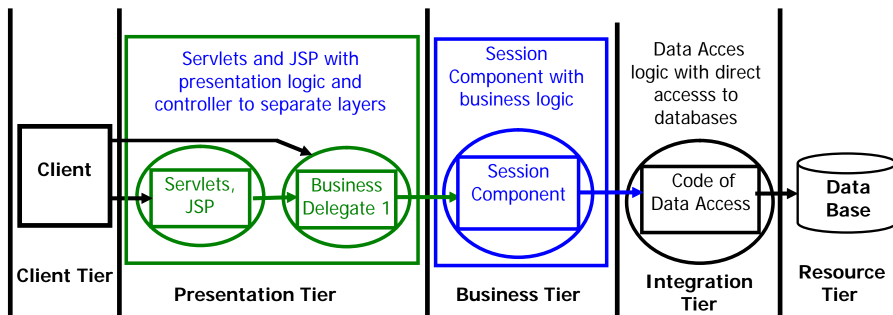


Figure 6.4. Architecture of information system with the Integration Tier [2].

Requirements:

– Our application should need to implement data access mechanisms to retrieve and modify data in the permanent storage,

– The persistent storage system should be separated from the rest of application,

– Our application should create an uniform interface to access data stored in various sources for example in RDBMS, LDAP, OODB, XML repository, flat file,

– Organization of the data access logic should be hidden in one location; specific functions of data access achieve greater portability and ease of maintenance code.

The *Data Access Object*:

– It is standardized and contains all code for access permanent data store,
– It manages connections to data sources to retrieve and save data,
– It is stateless,
– It does not cache data obtained during data collection,
– It encapsulates implementation details of the data store,
– We do not disclose exceptions, data structures, objects related to types of sql libraries.

Components of the pattern:

– *Client* – It is the object requesting access to data in order to download or save data. A client may be the business object (object model), the *SessionFacade* or *Application Service* component or any other object that requires secondary access to durable data,
– *DataAccessObject* – It is a key element of the pattern. It hides the actual implementation of data access from the client, to provide equal access, independent of the *DataSource*. This object implements the CRUD operations (Create, Read, Update, Delete),
– *DataSource* – It is any service data management. This may be a relational or object-oriented database, file system, repository, XML, etc.,
– *ResultSet* - It represents a result of a query. For JDBC drivers, this is an instance of type java.sql.ResultSet,
– *Date* - It represents a transfer facility, which is used to transfer data.

Implementation:

– Custom data access object,
– Data Access Factory,
– A collection of transfer objects,
– A buffered collection of lines,
– A collection of lines read-only,
– A Wrapper List (Wrapper Rowset List).

Properties:

– Hiding data access,
– Creation of object-oriented mechanisms for data access and concealment of database schemas,
– Facilitate data migration by replacing the DAO tier,
– Reducing the complexity of client code,
– Introducing an additional tier,
– Putting all the data access code inside a separate tier,
– Project requires a hierarchy of classes,
– Introducing of the complexity for the use of object-oriented data access methods.

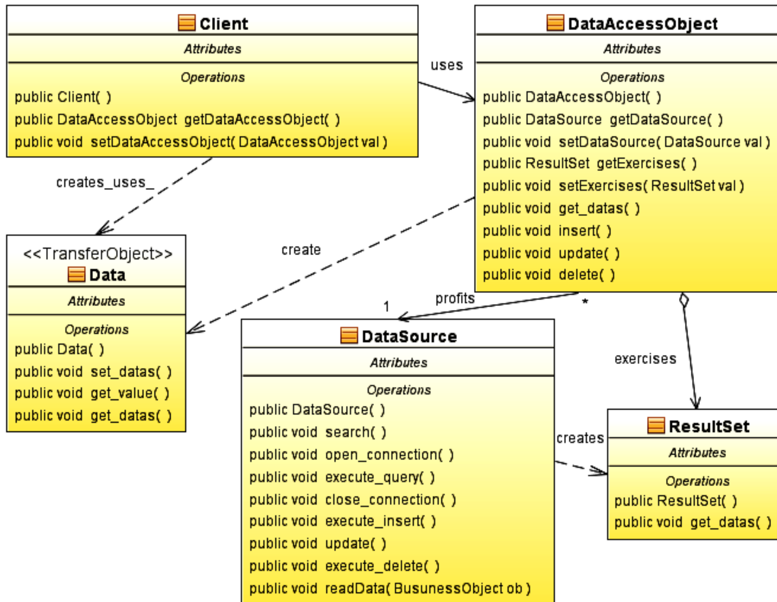


Figure 6.5. Class Diagram of the *Data Access Object* Pattern.

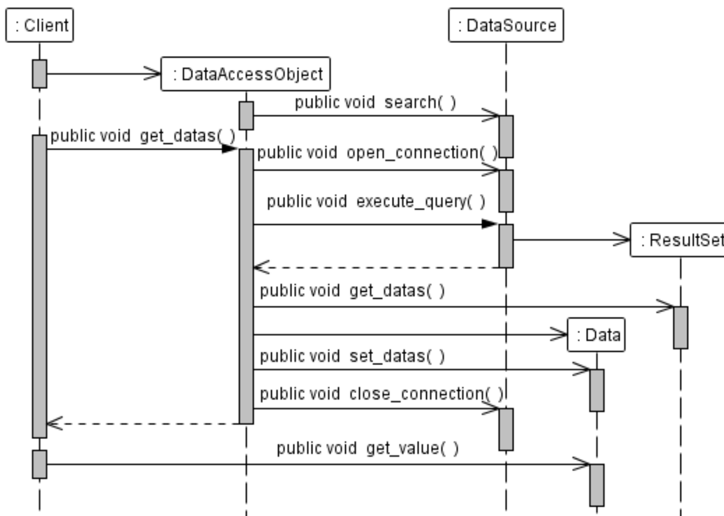


Figure 6.6. Sequence Diagram to download data using the *Data Access Object* Pattern.

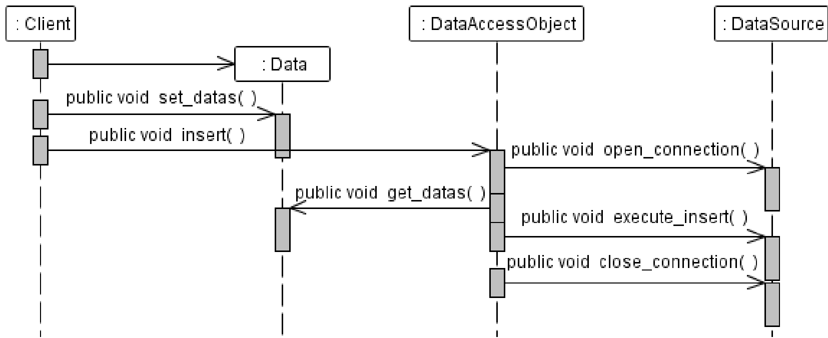


Figure 6.7. Sequence Diagram to insert data using the *Data Access Object* Pattern.

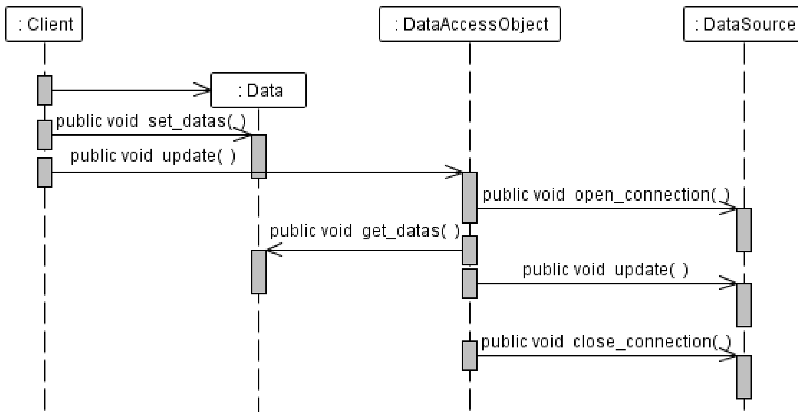


Figure 6.8. Sequence Diagram to update data using the *Data Access Object* Pattern.

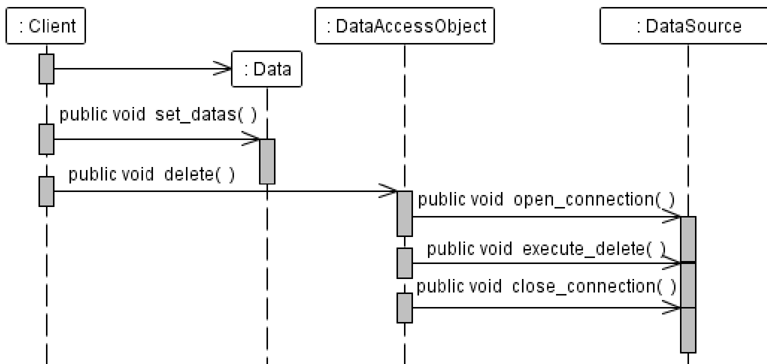


Figure 6.9. Sequence Diagram to delete data using the *Data Access Object* Pattern.

6.3.2. Domain Store

Problem 2 – Separation of persistence mechanisms from the object model:

– Many systems have a complex object model, built from the ordinary objects or components, such as *Entity*. They require a sophisticated strategy of preserving the permanent storage,

– Case 1 - container manages the persistence tier of business objects of the type *Entity*,
– Case 2 - the objects of *Entity* manage persistence, which can be used to implement the persistence object model,
– Case 3 – It runs applications in the Presentation Tier and separation between the persistence mechanism and an object model. Object models can be based on the so-called invisible life, that is, business objects of the object model do not correspond to the persistence mechanisms.

Possible solutions of the persistence mechanisms:

1. An object model does not exist (no objects of type *Entity*) - a solution to the problem 1 (solution 1).
2. The persistence mechanism is implemented in containers, which are objects of the *Entity* (strategy-CMP Container-Managed Persistence) - no inheritance in the object model (solution 2).
3. The persistence mechanism is implemented at the premises of the *Entity* (strategy BMP - Bean-Managed Persistence); code of persistence mixed with the object model - no inheritance in the object model (solution 3).
4. The mechanism of persistence is placed in the object model of the *Entity* - code of persistence mixed with the object model (solution 4).
5. The mechanism of persistence is separated from the object model of the *Entity* based on all object-oriented patterns (solution 5).

Figure 6.10 shows the model of the Business Tier based on 2, 3, 5 solutions of persistence.

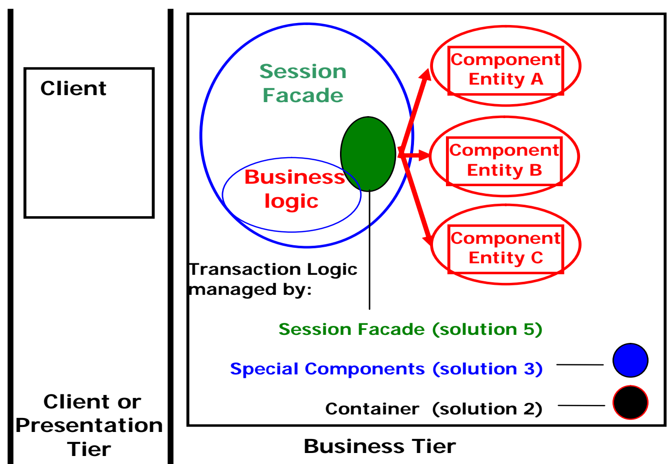


Figure 6.10. Business Tier based on 2, 3, 5 solutions of persistence [2].

Requirements:

– Implementation of persistence mechanisms in business objects such as *Entity* should be avoided,
– Objects of the type *Entity* that use the persistence mechanisms of the container should be refrained,
– The application can run in the web container,
– The object model uses inheritance and complex relationships,

– The mechanism of persistence can be solved in two ways:
  - make your own skeleton of persistence,
  - or use a ready-made solutions based on JDO or own O-R solutions.

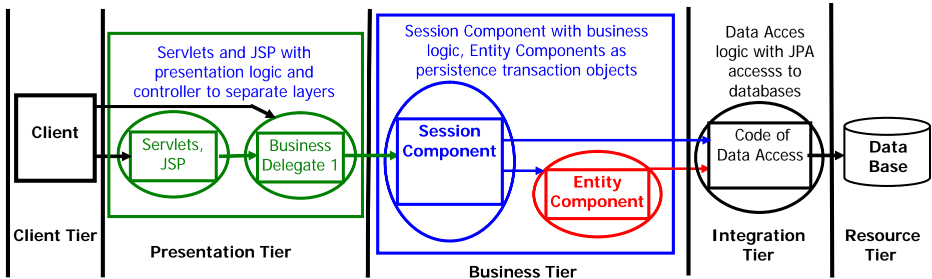


Figure 6.11. Architecture of 5-tiers information system [2].

The *Domain Store*:

– Provides an invisible persistence of an object model,

– Durability associated with the implementation of the *Domain Store* is separated from the object model (as opposed to container-managed persistence (CMP) or bean-managed persistence (BMP).

Components of the pattern:

– *Application Service* - It provides an object service using the object model,

– *Persistable* – It is an interface or a base class for all persistent business objects,

– *PersitenceManagerFactory* - It creates objects and manages the *PersistenceManager* component,

– *PersistenceManager* - It manages persistence and a query object model. *PersistenceManager* component manages objects via the object model together with the *StateManager* component,

– *StateManager* – It manages the state of the objects of the object model. It forces a transaction and retrieval of objects from the *DataResource* component,

– *StoreManager* (DAO) - It interacts with the DataResource to perform CRUD operations (Create, Read, Update, Delete). The *StoreManager DAO* component contains all the mechanisms to access data,

– *DataResource* – It is any service management data. This may be a relational or object-oriented database, etc.

Additional components of the pattern:

– *SessionFacade* – It is an access point to the Business Tier. It cooperates with one or more *ApplicatonService* objects,

– *PersistMap* – It contains definitions of relationships between objects and the mapping between the persistent objects and data source,

– *Transaction* – It is associated with the *PersistenceManager* object. It is used to define the rules of the transactions themselves and to manage transactions in environments without such support,

– *Query* – It encapsulates a query, the criteria for filtering, sorting, and parameter declaration.

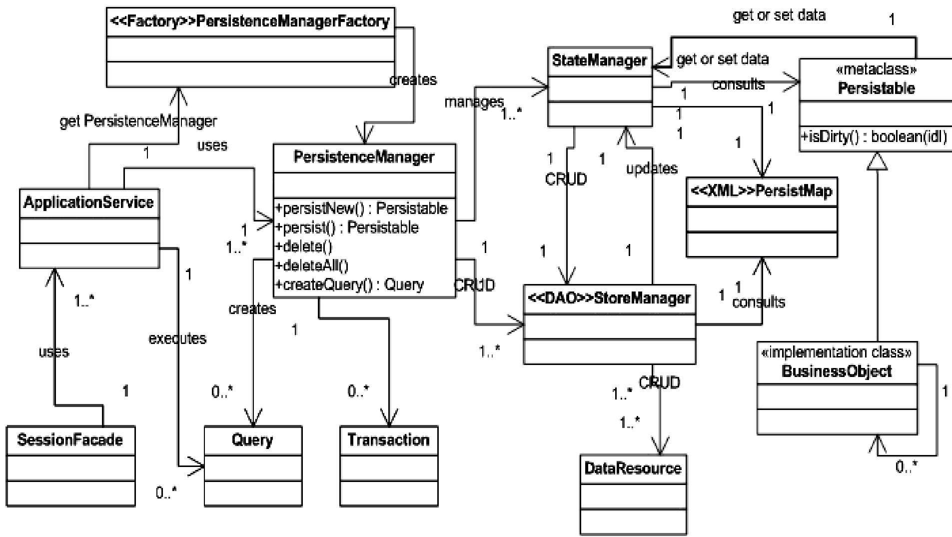


Figure 6.12. Class Diagram of the *Domain Store* of Pattern.

An algorithm of creating and persisting of objects:

1. The *ApplicationService* creates the *BusinessService*.
2. The *ApplicationService* retrieves the *PersistenceManager* from the *PersistenceManagerFactory*.
3. The *PersistenceManager* registers in the *TransactionManager*.
4. The *ApplicationService* asks the *PersistenceManager* to persist the *BusinessObject*.
5. The PersistenceManager creates the StateManager and instructs him to took BusinessObject.
6. The *StateManager* informs the *BusinessObject* that from now it manages its condition.
7. The *ApplicationService* asks the *PersistenceManager* to approve the deal explicitly or implicitly.
8. The *TransactionManager* informs the *PersistenceManager* to do empty all *StateManager* objects in one transaction.
9. The *PersistenceManager* informs the *StateManager* that it has to save own data.
10. The *StateManager* retrieves data from the *BusinessObject* object.
11. The *StateManager* orders the *StoreManager* to save data.
12. The transaction is confirmed.

Implementation:

– Own implementation of persistence,
– Downloading of the on-demand,
– JDO (Java Data Object).

Properties:

– Create own implementation of persistence is a complex task,
– Loading and saving multi-level tree of objects requires optimization,
– Allows you to better understand mechanisms of persistence principles,

– Advanced mechanisms for persistence may be too powerful for a simple object model,
– Facilitate testing of an object model,
– Separation of a business object model from the persistence logic.

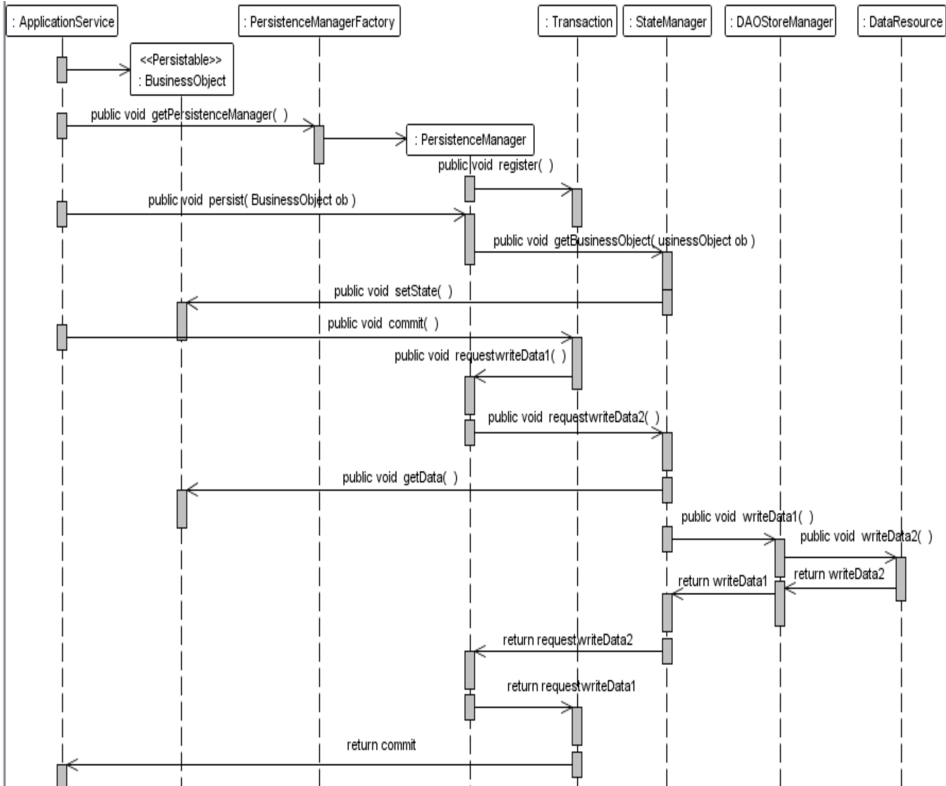


Figure 6.13. Sequence Diagram to create and persist business objects of the *Domain Store* Pattern.

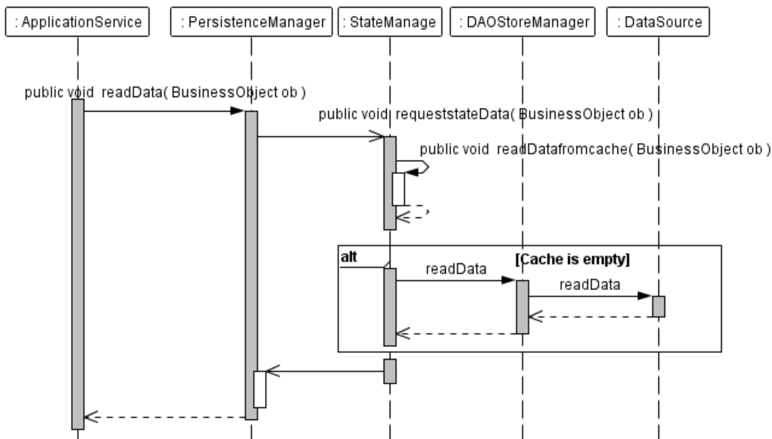


Figure 6.14. Sequence Diagram to download business object of the *Domain Store* Pattern.

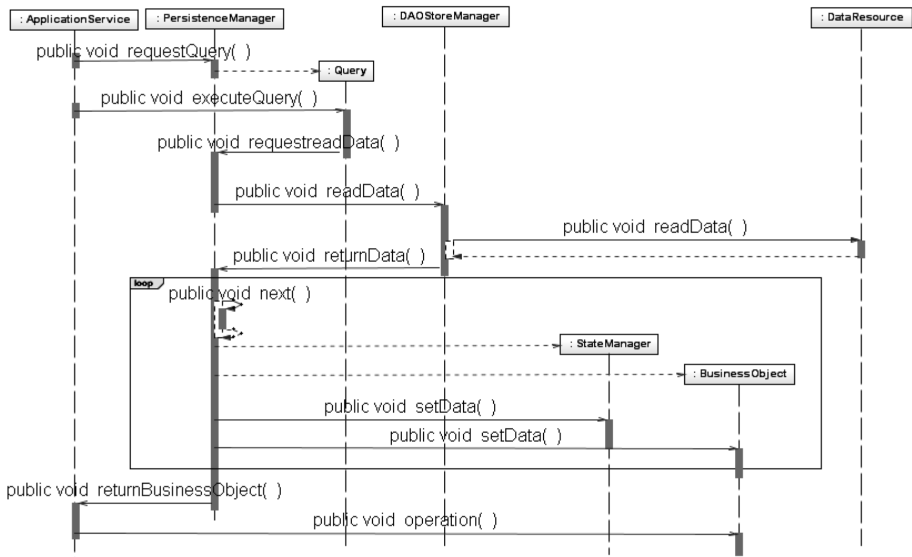


Figure 6.15. Sequence Diagram to create and perform the query business object of the *Domain Store* Pattern.

7. Example of the multitiered web application

7.1. Two examples of architectures of the multitier application as the *Visual Web Java Server Pages* applications

Figure 7.1 shows an example of the architecture of a *Web Application* based on synchronization of data by databases.

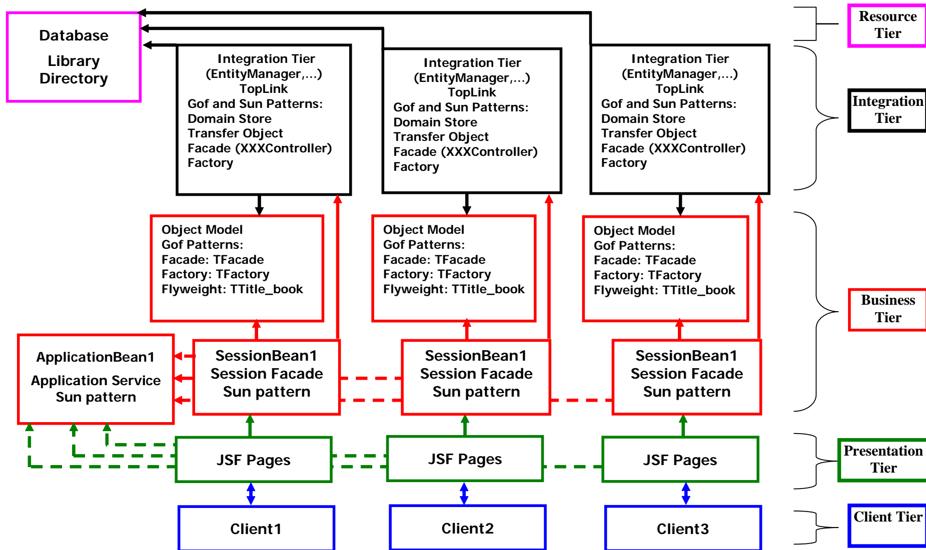


Figure 7.1. Architecture of the web application based on synchronization of data by databases.

Architecture of a *Web Application* based on synchronization of data by databases, when there are:

- Many clients of the Client Tier as the www pages,
- Many JSF pages of the Presentation Tier – each client has own pages,
- Many *Session Facade Components* of the Business Tier (*SessionBean1*) as remote facades of the *Business Service Sub-tier* (*Java Application* project) based on the POJO objects – each client has own *SessionBean1* and the own *Business Service Sub-tier,*
- Own *Domain Store Components* of the Integration Tier.

Figure 7.2 shows the example of the architecture of web application based on synchronization of data by application.

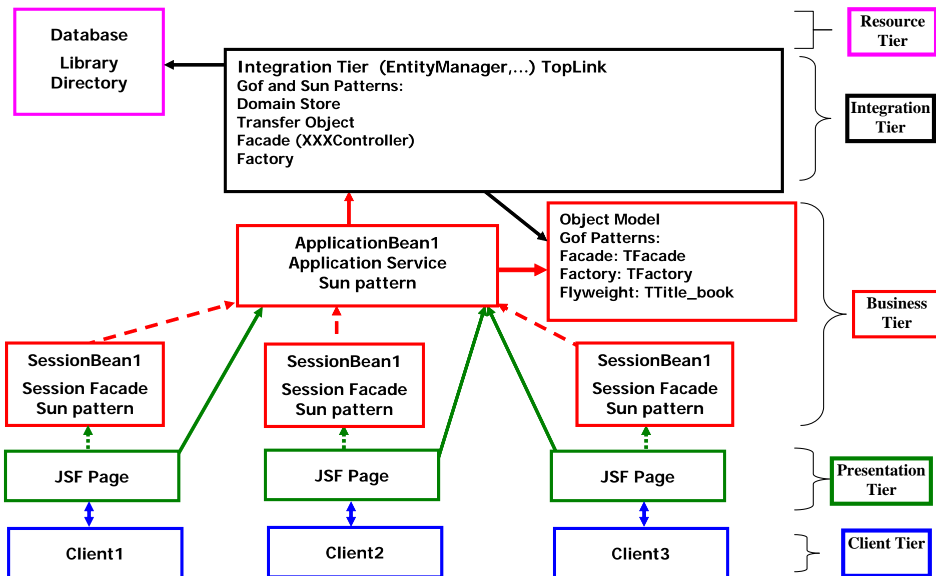


Figure 7.2. Example of the architecture of the web application based on synchronization of data by an application.

Architecture of a *Web Application* based on synchronization of data by application, when there are:

- Many clients of the Client Tier as the www pages,
- Many JSF pages of the Presentation Tier – each client has own pages,
- Common *Application Service Component* of the Business Tier (*ApplicationBean1*) as the remote facade of the *Business Service Sub-tier* (Java Application project) based on the POJO objects – each client uses the common *ApplicationBean1,*
- Common *Domain Store Components* of the Integration Tier.

7.2. The *Visual Web Java Server Pages* application based on synchronization of data by an application

Section 7.2 includes a few subsections, which illustrate the *Web Application* by using screenshots of this project from NetBeans 6.7.1 with the Visual Web JSF plugin.

7.2.1. Structure of project

Figure 7.2 shows structure of the *Enterprise Application* based on the *Java Application* project with Business and Integration Tiers and the *Web Application* project integrated with the previous project. The *Java Application* project includes packages, the first one representing the Integration Tier and the second one the *Service Sub-tier* as the part of the *Application Service* with the object model. The web project includes the Presentation Tier as the *JavaServer Faces Pages* (JSF Pages) and the remote part of the Business Tier as the *Application Service*.

Figure 7.3 presents a one of JSF Pages and components of remote part of the Business Tier as the *Application Service*: *RequetsBean1*, *SessionBean1,* and *ApplicationBean1* objects. Figure 7.4 shows some main classes of the integrated *Enterprise* project.

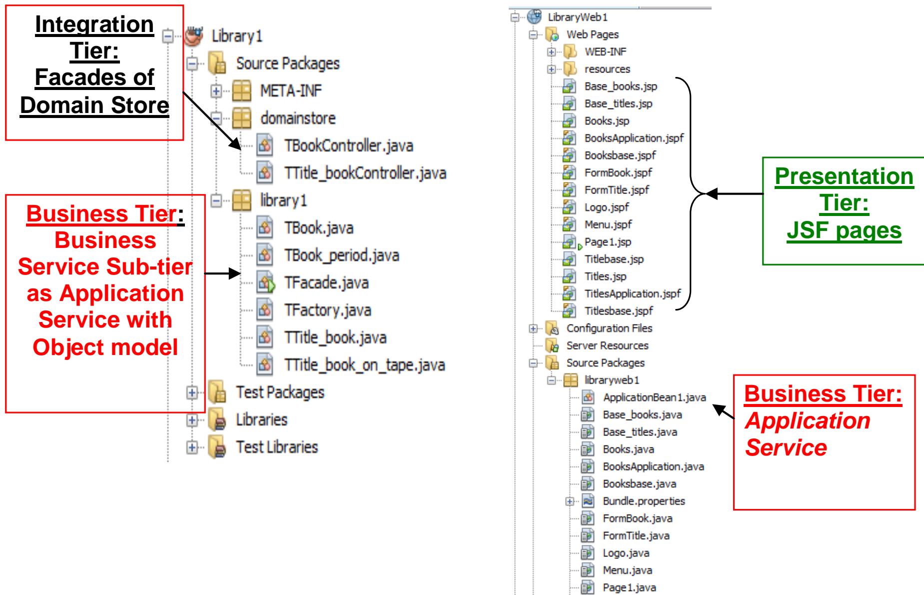


Figure 7.3. Structure of the *Enterprise Application* based on the *Java Application* Project with Business and Integration Tiers and the *Web Application* project integrated with the previous project.

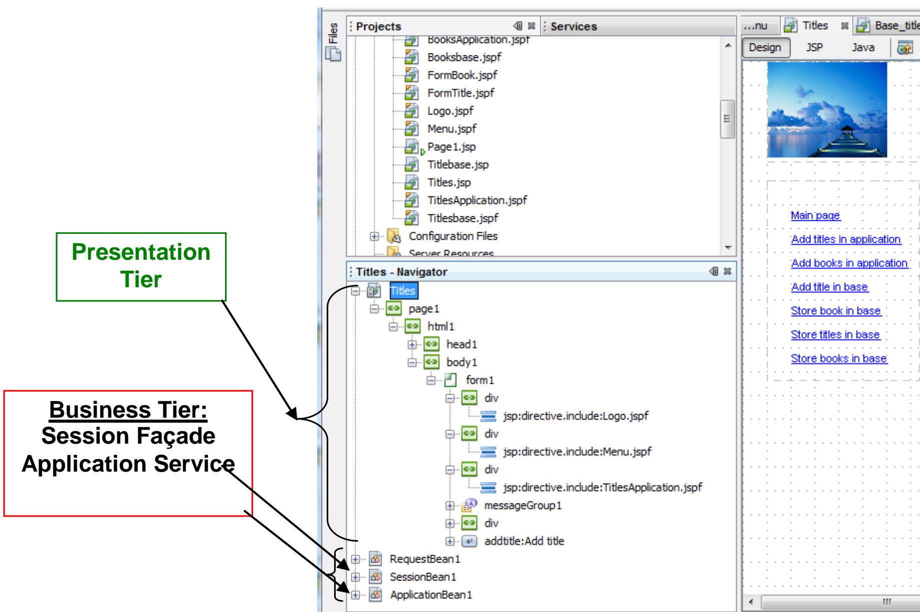


Figure 7.4. An example of one of JSF Pages and components of the remote part of the Business Tier.

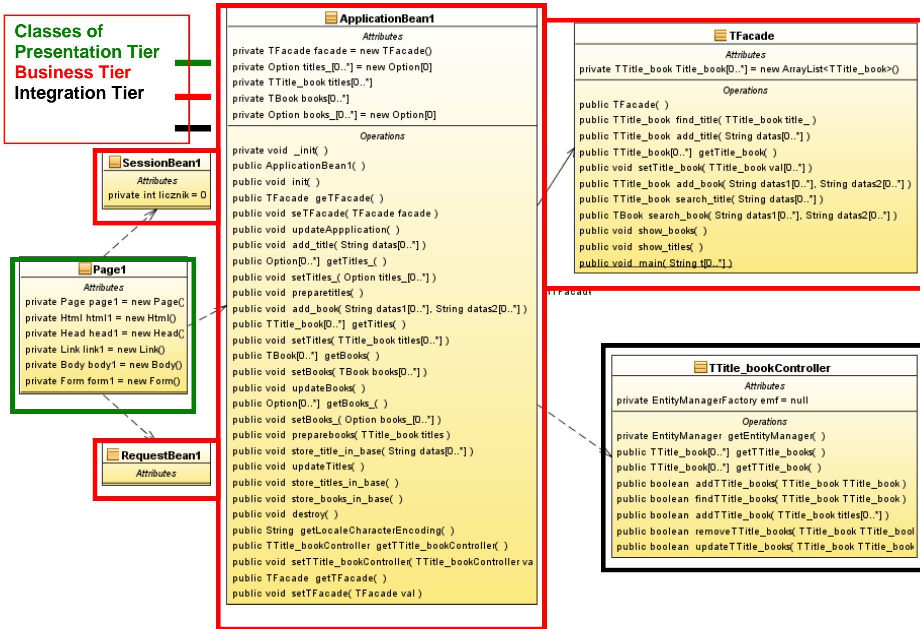


Figure 7.5. Some main classes of the integrated *Enterprise* Project, participating in individual tiers.

7.2.2. Business Service Sub-tier

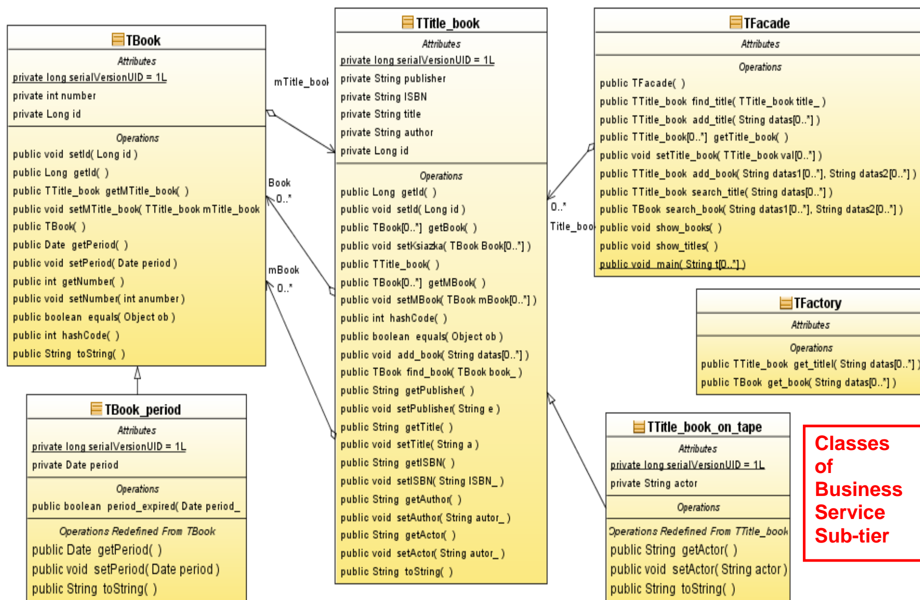


Figure 7.6. Class Diagram of the *Service Sub-tier* of the Business Tier.

```
...ava | TFacade.java | Page1 | Menu | Titles | Base_titles | ApplicationBean1.java *...
14
15  public class TFacade implements Serializable {
16
17      private ArrayList<TTitle_book> Title_book = new ArrayList<TTitle_book>();
18
19      public TFacade() {     }
20
21      public synchronized TTitle_book find_title(TTitle_book title_) {
22          int idx;
23          if ((idx = Title_book.indexOf(title_)) != -1) {
24              title_ = Title_book.get(idx);
25              return title_;  }
26          return null;
27      }
28
29      public synchronized TTitle_book add_title(String datas[]) {
30          TFactory factory = new TFactory();
31          TTitle_book title_book = factory.get_titlel(datas);
32          if (find_title(title_book) == null) {
33              Title_book.add(title_book);
34              return title_book;  }
35          return null;
36      }
```

Facade of
Business Service Sub-tier
as POJO class (1):
business logic methods
of sub-tier

Figure 7.7. Code of one of classes in the *Service sub-tier* of the Business Tier.

```
..ava | TFacade.java | Page1 | Menu | Titles | Base_titles | ApplicationBean1.java *...
44
45  public synchronized TTitle_book add_book(String datas1[], String datas2[]) {
46      TTitle_book help, help1 = null;
47      TFactory factory = new TFactory();
48      help = factory.get_titlel(datas1);
49      if ((help1 = find_title(help)) != null) {
50          help1.add_book(datas2);
51      }
52      return help1;
53  }
54
55  public synchronized TTitle_book search_title(String datas[]) {
56      TFactory factory = new TFactory();
57      TTitle_book title_book = factory.get_titlel(datas);
58      return find_title(title_book);
59  }
60
61  public synchronized TBook search_book(String datas1[], String datas2[]) {
62      TFactory factory = new TFactory();
63      TTitle_book title_book = factory.get_titlel(datas1);
64      TTitle_book title = find_title(title_book);
65      if (title != null) {
66          TBook book = factory.get_book(datas2);
67          return title.find_book(book);
68      }
69      return null;
70  }
```

Facade of
Business Service Sub-tier
as POJO class (2):
business logic methods of
sub-tier

Figure 7.8. Code of one of classes in the *Service sub-tier* of the Business Tier (related to Figure 7.6).

```
14
15  public class TFacade implements Serializable {
16
17      private ArrayList<TTitle_book> Title_book = new ArrayList<TTitle_book>();
18
19      public TFacade() {...}
20
21      public synchronized TTitle_book find_title(TTitle_book title_) {...}
28
29      public synchronized TTitle_book add_title(String datas[]) {...}
37
38      public synchronized ArrayList<TTitle_book> getTitle_book() {...}
40
41      public synchronized void setTitle_book(ArrayList<TTitle_book> val) {...}
44
45      public synchronized TTitle_book add_book(String datas1[], String datas2[]) {...}
54
55      public synchronized TTitle_book search_title(String datas[]) {...}
60
61      public synchronized TBook search_book(String datas1[], String datas2[]) {...}
71
72      public synchronized void show_books() {...}
86
87      public synchronized void show_titles() {...}
96
97      public static void main(String t[]) {...}
166 }
```

Facade of Business Service Sub-tier as POJO class (3): summarization of business logic

Figure 7.9. Code of one of classes in the *Service sub-tier* of the Business Tier (related to Figure 7.6).

```
1   package library1;
2   import java.io.Serializable;
3   import java.util.ArrayList;
4   import java.util.Collection;
5   import javax.persistence.Entity;
6   import javax.persistence.GeneratedValue;
7   import javax.persistence.GenerationType;
8   import javax.persistence.Id;
9   import javax.persistence.OneToMany;
10  import javax.persistence.Transient;
11
12  @Entity
13  public class TTitle_book implements Serializable {
14      private static final long serialVersionUID = 1L;
15      private String publisher;
16      private String ISBN;
17      private String title;
18      private String author;
19
20      @Id
21      @GeneratedValue(strategy = GenerationType.AUTO)
22      private Long id;
23      public Long getId()          {  return id;     }
24      public void setId(Long id)   {  this.id = id;  }
25
26      @OneToMany(mappedBy = "mTitle_book")
27      private Collection<TBook> Book;
28      public Collection<TBook> getBook()               {  return Book;      }
29      public void setKsiazka(Collection<TBook> Book)   {  this.Book = Book;  }
30
31      public TTitle_book()                             {  id=null;        }
```

Example of Entity class from Object Model of Business Service Sub-tier (1):

Persistence Annotation

Figure 7.10. Code of one of classes in the *Service sub-tier* of the Business Tier (related to Figure 7.6).

```
@Transient
private ArrayList<TBook> mBook = new java.util.ArrayList<TBook>();
public ArrayList<TBook> getMBook()              { return mBook; }
public void setMBook(ArrayList<TBook> mBook){ this.mBook = mBook; }

@Override
public int hashCode() { return (id != null ? id.hashCode() : 0); }

@Override
public boolean equals(Object ob) {
    boolean a = false;
    if (getISBN().equals(((TTitle_book) ob).getISBN()))
        if (getActor().equals(((TTitle_book) ob).getActor()))
            a = true;
    return a;  }

public void add_book(String datas[]) {
    TFactory factory = new TFactory();
    TBook anew;
    anew = factory.get_book(datas);
    if (find_book(anew) == null) {
        mBook.add(anew);
        anew.setMTitle_book(this);  }
}

public TBook find_book(TBook book_) {
    int idx;
    if ((idx = mBook.indexOf(book_)) != -1) {
        book_ = (TBook) mBook.get(idx);
        return book_; }
    return null;
}
```

Example of Entity class from Object Model of Business Service Sub-tier (2):

business logic methods

Figure 7.11. Code of one of classes in the *Service sub-tier* of the Business Tier (related to Figure 7.6).

7.2.3. Application Service of the Business Tier as the remote sub-tier

```
public class ApplicationBean1 extends AbstractApplicationBean {
    Managed Component Definition
    /**...*/
    public ApplicationBean1() {
    }
    /**...*/
    @Override
    public void init() { // Perform initializations inherited from our
        super.init();
        // Perform application initialization that must complete
        // *before* managed components are initialized
        // TODO - add your own initialiation code here
        Managed Component Initialization
        // *after* managed components are initialized
        // TODO - add your own initialization code here
        updateTitles();
        updateBooks();
        updateAppplication();
        preparetitles();
    }
    private TFacade facade = new TFacade();

    public TFacade geTFacade()              { return facade;    }

    public void seTFacade(TFacade facade)   { this.facade = facade;  }
```

Figure 7.12. Code of one of classes in the remote sub-tier of the Business Tier (related to Figure 7.6).

```
 90     public void updateAppplication() {...}
107
108     public void add_title(String datas[]) {
109         getTFacade().add_title(datas);
110     }
111     private Option titles_[] = new Option[0];
112
113     public Option[] getTitles_()            {  return titles_;           }
114
115     public void setTitles_(Option[] titles_) {  this.titles_ = titles_;   }
116
117     public void preparetitles() {
118         ArrayList<TTitle_book> atitles = facade.getTitle_book();
119         int amount = atitles.size();
120         if (amount > 0) {
121             Option help[] = new Option[amount];
122             Iterator iterator = atitles.iterator();
123             int i = 0;
124             while (iterator.hasNext()) {
125                 help[i++] =
126                         new Option(Integer.toString(i), iterator.next().toString());
127             }
128             titles_ = help;
129         }
130     }
```

remote method for methods of Business Service Sub-tier

binding data of JSF Component

Figure 7.13. Code of one of classes in the remote sub-tier of the Business Tier (related to Figure 7.6).

7.2.4. Integration Tier

```
212     public void store_titles_in_base() {
213         // Add the new Entity to the database using UserController
214         TTitle_bookController Title_bookController =
215                 new TTitle_bookController();
216         Title_bookController.addTTitle_book(getTFacade().getTitle_book());
217     }
218
219     public void store_books_in_base() {
220         // Add the new Entity to the database us
221         TBookController bookController =
222                 new TBookController();
223         bookController.addTBooks_(getTFacade().getTitle_book());
224     }
```

remote methods for methods of Integration Tier

Figure 7.14. Code of one of classes in the remote sub-tier of the Integrated Tier (related to Figure 7.6).

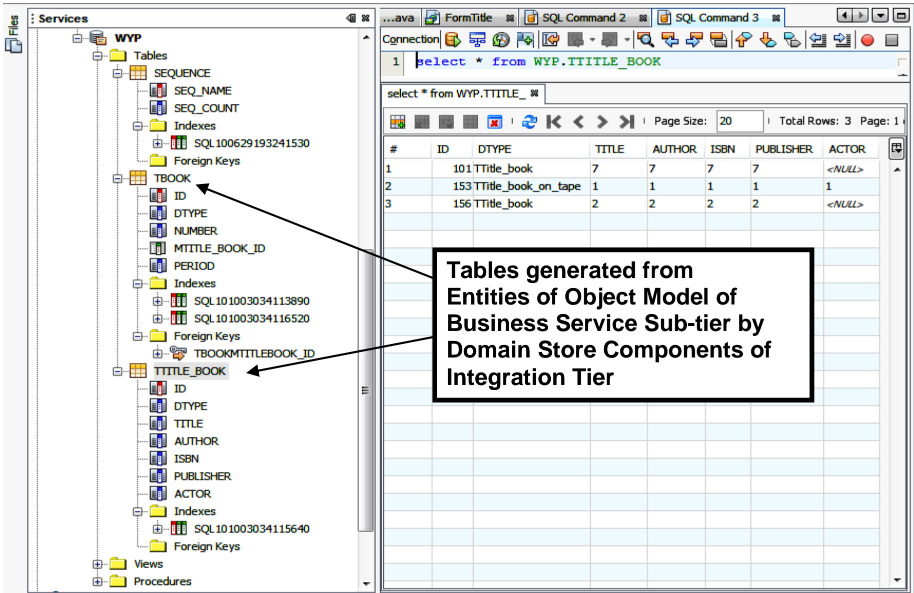


Figure 7.15. Tables generated from data model of the Integrated Tier. Data model represents the Object model with annotations of the Persistence Java language.

7.2.5. Presentation Tier

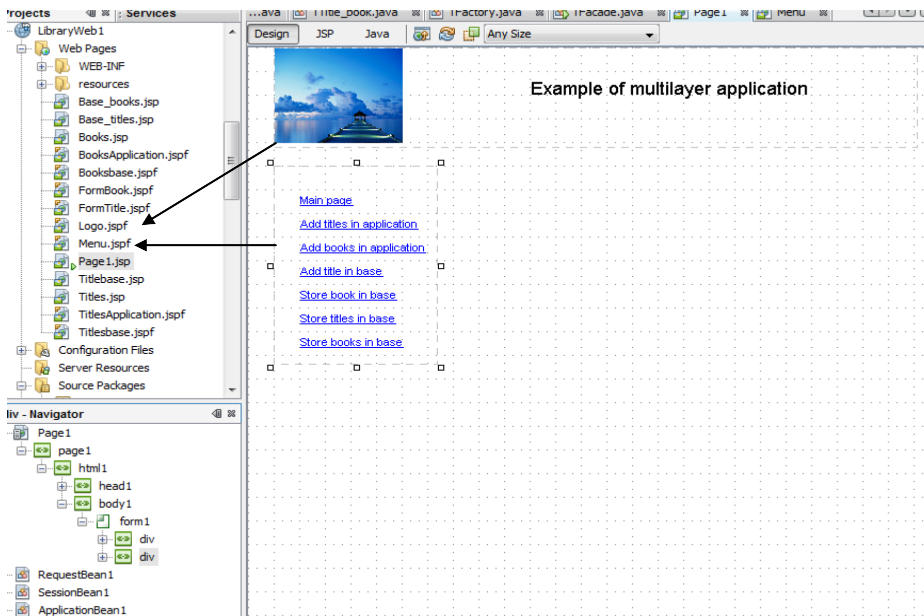


Figure 7.16. One of JSF Pages of the Presentation Tier, related to Figure 7.6 – the main view form consists of some reusable sub-views as the *Fragment Box* components.

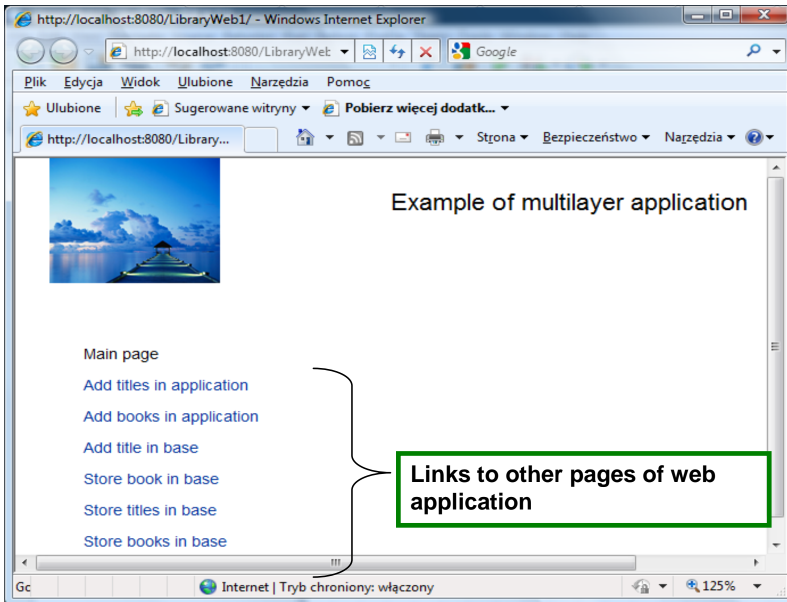


Figure 7.17. One of JSF Pages of the Presentation Tier (related to Figure 7.6).

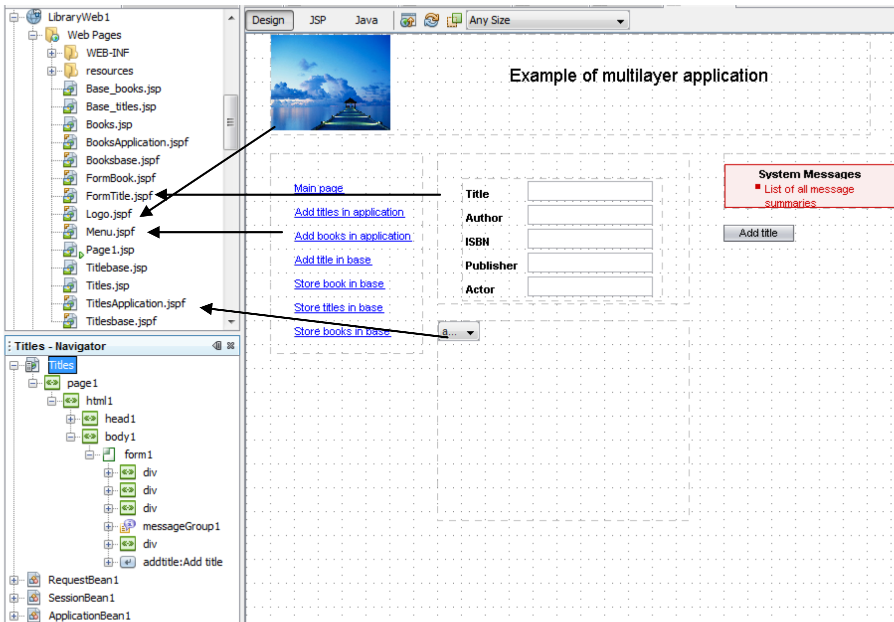


Figure 7.18. One of JSF Pages of the Presentation Tier (related to Figure 7.6) – the main view form consists of some reusable sub-views as the *Fragment Box* components.

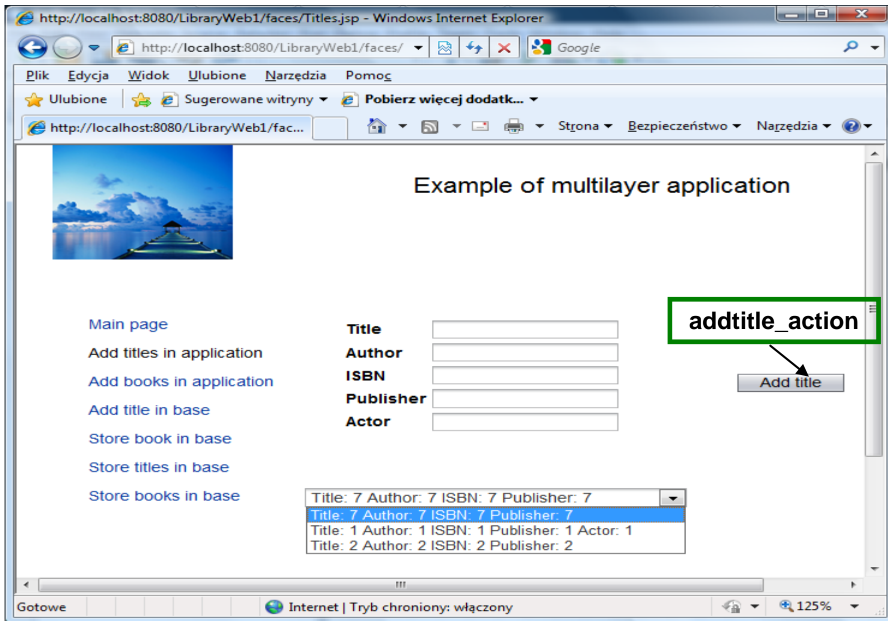


Figure 7.19. View of one of JSF Pages of the Presentation Tier, related to Figure 7.18.

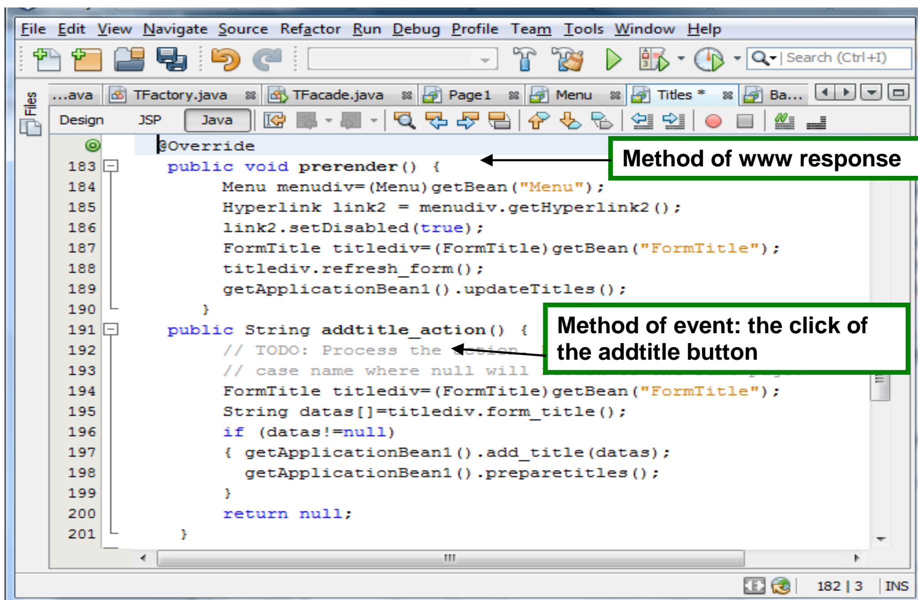


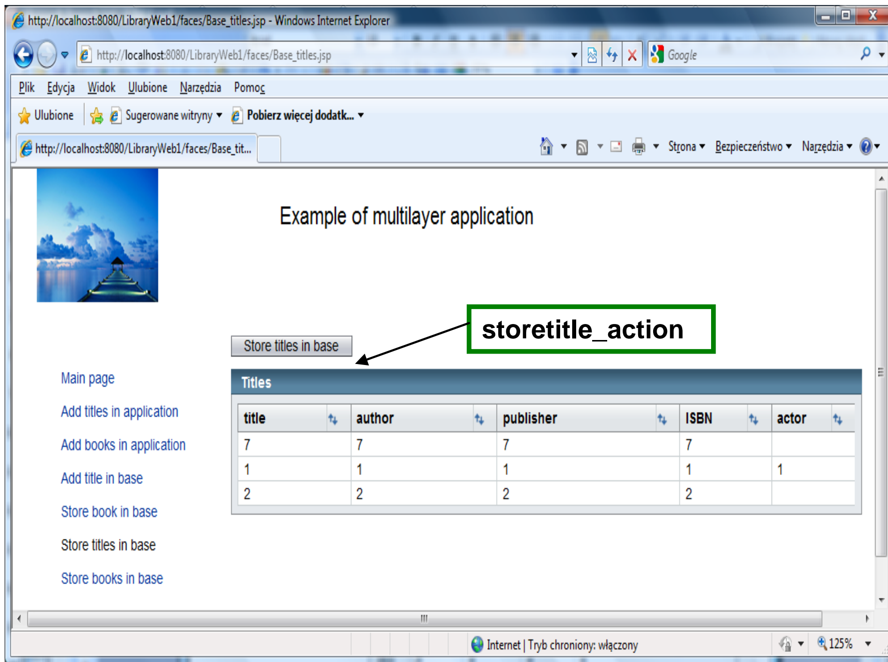
Figure 7.20. Code handling the event of one of JSF Pages of the Presentation Tier (related to Figure 7.6) – calling methods from the sub-view of the main form view for input data and from the remote sub-tier of the Business Tier for executing the called service by the *Service Sub-tier*.

```
211 public void refresh_form()
212 {    titleapplicationpanel.setRendered(true);
213      title.setText("");
214      author.setText("");
215      ISBN.setText("");
216      publisher.setText("");
217      actor.setText("");
218 }
219 public String [] form_title()
220 {
221    String what;
222    if ((author.getText().equals("")||title.getText().equals("")||
223       ISBN.getText().equals("")||publisher.getText().equals("")))
224    return null;
225    if (actor.getText().equals("")) {
226        what = "1";
227    } else {
228        what = "3";
229    }
230    String datas[] = {what, (String) author.getText(),
231     (String) title.getText(), (String) ISBN.getText(),
232     (String) publisher.getText(), (String) actor.getText()};
233    return datas;
234 }
```

Methods of the FragmentBox Component as the sub-view of the Titles form view

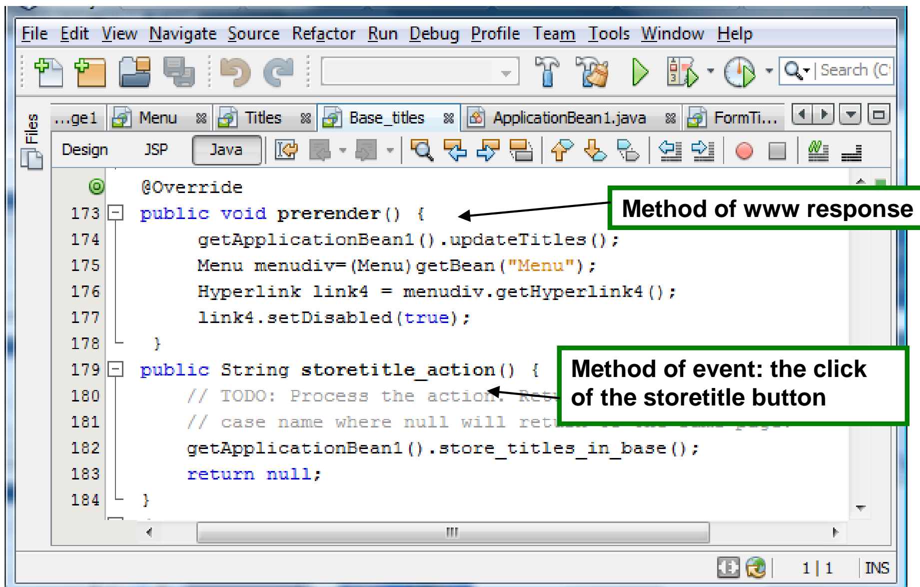
Figure 7.21. Method handling the event called from the sub-view of the main form view.

Figure 7.22. One of JSF Pages of the Presentation Tier (related to Figure 7.6) – the main view form consists of some reusable sub-views as the *Fragment Box* components.



title	author	publisher	ISBN	actor
7	7	7	7	
1	1	1	1	1
2	2	2	2	

Figure 7.23. View of one of JSF Pages of the Presentation Tier, related to the Figure 7.22.




```
    @Override
173 public void prerender() {
174         getApplicationBean1().updateTitles();
175         Menu menudiv=(Menu)getBean("Menu");
176         Hyperlink link4 = menudiv.getHyperlink4();
177         link4.setDisabled(true);
178     }
179 public String storetitle_action() {
180       // TODO: Process the action. Ret
181       // case name where null will ret
182       getApplicationBean1().store_titles_in_base();
183       return null;
184   }
```

Figure 7.24. Code of the handling the event of one of JSF Pages of the Presentation Tier (related to Figure 7.6) – calling methods from the remote sub-tier of the Business Tier for executing the called database service by components of the Integration Tier.

PART II

XML-based service description languages

8. RDF (Resource Description Framework)

The Resource Description Framework (RDF) is an open-world framework for representing information about resources in a graph form. The framework, primarily intended for representing metadata about WWW resources, makes use of Uniform Resource Identifier (URI) or IRI (Internationalized Resource Identifier) to name or to refer to a particular resource.

A concept of a resource has a long history. At the beginning, a term resource was associated with a document or file, which was available in a computer network under a statically assigned address. The development of Internet technology has upgraded this definition. Today's resource can be anything uniquely recognizable in the computer network systems by the name, address, credentials or any other identifier. Therefore, the RDF, which uses URI or IRI, is a language, which theoretically allows describing anything clearly named or identified.

The use of resource identifiers is a mapping, which indicates the actual entities or objects. However, this mapping may have a broader context. Identifiers attribute not only real things, but also concepts, ideas or any other abstracts. Most often URI or IRI identifiers represent addresses where the resources are located at, and where they can be downloaded from. However, the resources, in general, do not have to be accessible on the Web. Moreover, it is not required that complete information about them is available.

URI and IRI identifiers can refer to anything and, by the open-world assumption, anyone can make statements about any resource. There are no restrictions preventing anyone from making assertions, which are nonsensical or inconsistent with other statements. More over, the identifiers interpretation can be context-dependent, as, for example, a link to a daily news web page. Thus, application designers working with RDF need to be aware about these issues.

NOTE:

The open-world assumption: "The truth of a statement is independent of whether it is known. In other words, not knowing that a statement is explicitly true does not imply that the statement is false".

The closed-world assumption: "Any statement that is not known to be true can be assumed to be false".

The no unique names assumption: "Unless explicitly stated otherwise, it cannot be assumed that resources that are identified by different URIs are different".

W3C organization maintains the RDF specification in the scope of W3C's Semantic Web Activity. RDF became a W3C Recommendation on 10 February 2004. The documents contributing to this recommendation [30, 33-37] are accessible from the W3C's web page.

RDF was designed to be read and understood by computers. The RDF/XML is a normative syntax for expressing RDF information in a computer-readable form. Apart from RDF/XML, there exist other languages for the RDF graph serialization, as Notation 3 (N3) and Terse RDF Triple Language (TURTLE). The syntax of those languages is less verbose than RDF/XML. Therefore, they are used to represent RDF graphs for humans. The N3 language is designed as a readable language for data on the Web that goes beyond RDF (it contains logical extensions and rules). The TURTLE is a RDF-only subset of N3.

NOTE:

URI – is a sequence of characters that identify a name or a resource on the Internet. This sequence is built from a limited subset of the repertoire of US-ASCII (ASCII) characters and consists of four parts:

`scheme ":" hier_part ["?" query ] [ # fragment ]`

where:

`scheme` – a string that begins with a letter followed by any combination of letters, digits, and the plus ("+"), period ("."), or hyphen ("-") characters.

`hier_part` – a string that begins with a double forward slash ("//"), followed by an `authority` part and an optional `path` intended to hold identification information hierarchical in nature.

`authority` - holds an optional user information part, which might include scheme-specific information about how to gain authorization to access the resource, terminated with "@", a hostname, and an optional port number preceded by a colon ":".

`path` - is a sequence of segments separated by a forward slash ("/") representing a hierarchy similar to the directory structure. Each segment can contain parameters separated from it using a semicolon (";"), though this is rarely used in practice.

`query` - is an optional part separated with a question mark, which contains additional identification information which is not hierarchical in nature. The query string syntax is not generically defined, but is commonly organized as a sequence of `<key>=<value>` pairs separated by a semicolon or separated by an ampersand, for example: `key1=value1;key2=value2` or `key1=value1&key2=value2`

`fragment` - is an optional part separated from the front parts by a hash ("#"). It holds additional identifying information that provides direction to a secondary resource, e.g. a section heading in an article identified by the remainder of the URI.

URI can appear in an absolute or relative form. The absolute form is a form, in which a resource is identified with full and context independent resource reference. Relative form is a form, in which resource reference has not given full information to identify a resource and missing information must be derived from the context. The URI, which contains relative part is URI reference. A URI reference (or URIref) is a URI, together with an optional fragment identifier at the end. For example `#section2` (relative URI) inside the document `http://www.example.org/index.html` corresponds to the absolute URI `http://www.example.org/index.html#section2`. The syntax of URI has been defined in RFC2396 [38] and updated in RFC2732 [40] – both obsolete, but widely implemented versions of the generic URI syntax. The current generic URI syntax specification is RFC3986 [41].

IRI – is an extension of URI providing much wider repertoire of characters allowed. Its definition is similar to URI. However, the class of unreserved characters has been extended by the characters of the Universal Character Set (UCS) [44] beyond U+007F, with the restriction that private UCS characters can occur only in query parts.

The definition of IRI is provided in RFC3987 [42], the specification that defines "internationalized" versions corresponding to other constructs from RFC3986 [41], such as URI references. In many cases URI and IRI are used interchangeably, but practical replacement of URIs (or URI references) by IRIs (or IRI references) depends on the application.

Some domain names appearing in the URL authority part have been reserved for testing or other similar uses as described in RFC2606 [39] document. Therefore, URLs with `example.com`, `example.net`, or `example.org` do not refer to any existing resources, but serve well for illustrative purposes in documentation.

8.1. Model

The RDF is built on an abstract concept of *Graph data model*. The basic element of this model is a *statement*, represented by a *triple*: <*subject, predicate, object*>. The triple links one object (*subject*) to another object (*object*) or a literal via a property (*predicate*). In another words: a resource (*subject*) has a property (*predicate*) valued by property value (*object*).

The RDF graph is a directed graph, in which *subjects* and *objects* are nodes and *predicates* are arcs, starting at a proper *subject* and leading to a proper *object*. Thus, the simplest RDF graph consists of just two nodes and one arc – a single triple representation. Because any *predicate* appearing in a triple can be also considered as a resource, it can be used in a recursive way as a subject of some other triple. Hence, given more triples, the graph can grow. For example, the same person can be a subject of a bunch of triples linking this person with various objects via number of properties, which in turn, can be associated with other resources.

What makes RDF triples special is that every part of a triple can have URI associated with it (and identifying it). Being more precise, RDF requires that *subject* has URI or is *b-node*; *predicate* has URI; and that *object* has URI, is *b-node* or is literal. Moreover, the same URI can be assigned to a node and to an arc as well.

Visualization of RDF graph is quite simple. Nodes of a graph (subjects and objects) are visualized as ovals, linked by directed arcs (predicates). However, there is one exception – a node that contains literal (what applies only to objects of triples) is visualized as a rectangle (see Figure 8.1).

Figure 8.1. Graphical representation of an RDF triple a) subject and object nodes are resources, b) object is literal.

RDF model intrinsically supports binary relations only; that is, a statement specifies a relation between two resources. However, the higher arity relations can be easily modelled. RDF supports also the concepts of containers and collections, which, used together, allow complex graph definitions (see Figure 8.2).

NOTE:

Representing information involving higher arity relations (relations between more than two resources) in RDF needs some effort. The common way to represent any n-ary relation in RDF is following. At first a subject of the original relation is selected. Then intermediate resource is defined to represent the rest of the relation (either with assigned URI or without it as *b-node*). Next, to represents the remaining components of the relation some new properties are assigned to this intermediate resource.

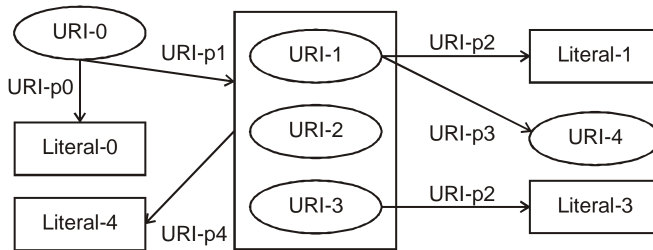


Figure 8.2. RDF graph with several interconnected subjects and objects, including container with three subjects grouped in one container (shown as a rectangle with three object nodes inside). The containers are often represented as star-like sub-graphs with blank nodes placed in the stars' origins.

An RDF model is very simple and uniform. Because URIs are defined in a single universal namespace, merging different RDF graphs is extremely easy – the triples can always be merged without name translations, so entire graphs can be transported and combined without any translation. This feature is highly appreciated by the developers of the web applications. On the other hand, the flexibility and extensibility of RDF model might cause some troubles in the knowledge-based systems and inference engines (for example, the inference might not be possible for a given graph). An example of an RDF graph representing a statement "John Smith is a father of Susan Smith" is shown in Figure 8.3.

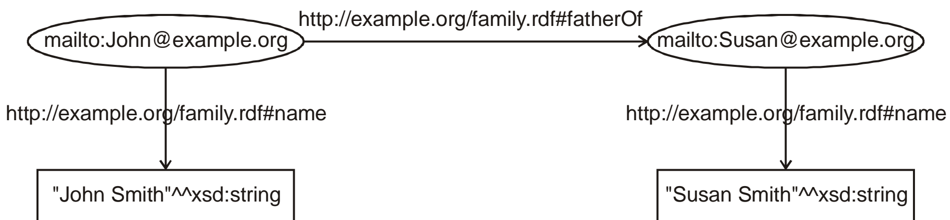


Figure 8.3. RDF graph representing a statement "John Smith is a father of Susan Smith". Both persons are identified by their e-mail addresses and their names are provided as typed literals.

NOTE

Literal in the RDF sense is a constant string value such as string or number. Literals cannot be the subjects of statements, only objects (target nodes in the RDF graphs). Literals can be either plain literals (without type) or typed literals typed using XML Datatypes. All literals have a lexical form being a Unicode string [44], which should be in Normal Form C. Plain literals can have an optional language tag assigned in form of a suffix starting with @ followed by the language code string (as defined by RFC3066, normalized to lowercase). Typed literals have a lexical form ending with a suffix being RDF datatype URI reference (as in XML Schema Datatypes or an URI of custom datatype defined). The suffix starts with two caret characters ^^.

Resource in the RDF sense can represent anything that can be named: an object, act, or concept. URIs identifies resources. Regarding RDF triples, resources can be subjects, predicates or objects.

A *blank node* (abbreviated with *b-node*) is a node of some RDF graph, which is neither identified by a URI nor not a literal. Such a node can be viewed as a graph scoped identifier that cannot be directly referenced from outside. It is used mainly for graph branching as for representing higher arity relations. In general, using *b-nodes* may cause some troubles in

different graphs merging or querying. In particular, the problem arises from possible node ID conflicts (merging) or temporary node ID assignments (querying). A *b-node* can be used in any RDF triple only as a subject or an object and cannot be used as a predicate (in in some syntaxes like Notation 3 it is acceptable to use a Blank Node as a predicate [12].

NOTE:

There is no built-in concept of numbers or dates or other common values in RDF. RDF predefines just one datatype: `rdf:XMLLiteral`, which is used for embedding XML in RDF. There is no mechanism for defining new datatypes as well. It is expected that any new datatypes will be provided separately, and identified with URI references, as XML Schema datatypes defined in [60]. Thus, the datatype abstraction used in RDF is compatible with the abstraction used in XML Schema. A datatype consists of a lexical space (a non-empty set of character strings), a value space (a non-empty set) and a lexical-to-value mapping (a mapping from the lexical space to the value space). For example, the lexical-to-value mapping for the XML Schema datatype `xsd:boolean`, where each member of the value space (represented here as 'T' and 'F') has two lexical representations, is as follows:

value space: `{T,F}`;

lexical space: `{"0", "1", "true", "false"}`;

lexical-to-value mapping: `{<"true", T>, <"1", T>, <"0", F>, <"false", F>}`.

Not all XML Schema datatypes are suitable for the use in RDF. `xsd:duration` – does not have a well-defined value space; `xsd:QName` and `xsd:ENTITY` – require an enclosing XML document context; `xsd:ID` and `xsd:IDREF` – are for cross-references within an XML document; `xsd:NOTATION` – is not intended for a direct use; `xsd:IDREFS`, `xsd:ENTITIES` and `xsd:NMTOKENS` – are sequence-valued datatypes which do not fit the RDF datatype model.

8.2. Vocabulary

The RDF is a language built on a set of URIs. There are two vocabularies defined in the specification: RDF and RDF Schema (RDFS).

The RDF vocabulary contains basic terms (Table 8.1) for expressing simple statements about resources using named properties and values [30].

The RDFS vocabulary is a semantic extension of the RDF vocabulary (Table 8.2) providing mechanisms for describing properties and relationships between these properties and other resources [37]. It allows custom vocabularies creation by introducing new terms using built-in vocabulary or terms already defined.

RDFS defines classes and properties that may be used to describe classes, properties and other resources. It includes terms that may be used to determine characteristics of other resources, such as domains (to indicate that the resource is of a particular RDF class) and ranges (to indicate that the resource is of a specific data type) of properties. Such description can be recursive. It is said that RDF Schema provides a type system for RDF.

The RDF and RDFS vocabularies can describe relationships between items from multiple vocabularies developed independently. To simplify such description the prefix strings corresponding to the XML namespace names are often used.

The concepts of classes and properties are similar to the concepts used in object-oriented programming. By convention, class's names start with an upper case letter, and properties

names – with a lower case letter (after the leading prefix). However, there is one important difference. "Instead of defining a class in terms of the properties its instances may have, the RDF vocabulary description language describes properties in terms of the classes of resource to which they apply". In object oriented-programming languages class definition implies the characteristic of instances, while in RDF properties definitions imply the class membership of the instance – if an instance has a certain property asserted with a domain defined, this domain specifies the class of this instance. This interpretation provides the basis for inference.

8.2.1. RDF vocabulary

The RDF vocabulary consists of some basic concepts for describing RDF triples, and concepts for describing containers and collections (gathered up in Table 8.1). The URI references of these concepts start with a leading substring `http://www.w3.org/1999/02/22-rdf-syntax-ns#`. For the sake of clarity, this substring is often substituted by a namespace prefix `rdf:`. Thanks to that, names of the concepts can be shortened, as, for example, `rdf:type`, whose full URI is `http://www.w3.org/1999/02/22-rdf-syntax-ns#type`.

RDF allows describing every element of RDF in RDF itself. It is done through RDF reification – disassembling triples into parts, which can be used then in other statement declarations, as a whole or as parts (RDF reification vocabulary).

The RDF containers and collections allow to describe groups of things. The main difference between these two kinds of description relies on the way of stating out that no more members of a group are present. The core RDF specifications define no mechanism for containers to determine this fact. Therefore, containers are open in that sense. On the contrary, collections cardinalities are determined. Collections are lists of items in terms of head-tail links. Thus, the number of items in a given list delimits the existence of explicit list's terminator. Thanks to such syntax, collections are not altered by RDF graphs merging, and differ from containers in allowing branching structures.

Table 8.1. RDF vocabulary.

RDF property and type vocabulary	RDF reification vocabulary	RDF container vocabulary	RDF collection vocabulary

`rdf:Property`
`rdf:type`
`rdf:XMLLiteral`
`rdf:value`
	`rdf:Statement`
	`rdf:subject`
	`rdf:predicate`
	`rdf:object`
		`rdf:Seq`
		`rdf:Bag`
		`rdf:Alt`
		`rdf:_1`
		`rdf:_2`
		`...`
			`rdf:List`
			`rdf:first`
			`rdf:rest`
			`rdf:nil`

`rdf:Property` – is a property (or, more formally, a class of RDF properties) used for the definition of predicates in triples. Each definition of a property might include restrictions regarding domain and range (using concepts from RDFS vocabulary). Even though properties are classes, they are defined and used independently of RDFS classes (defined with `rdfs:Class` from RDFS vocabulary).

`rdf:type` – is a property (or, more formally, an instance of `rdf:Property`) used to assert a type to a resource. The value of this property is a URI identifying a class (or, more formally, an instance of `rdfs:Class` defined with RDFS vocabulary). A triple of the form: `R rdf:type C` states that `C` is an instance of `rdfs:Class` and `R` is an instance of `C`. Asserting the same type to several resources is possible, as asserting any other predicates.

rdf:XMLLiteral - is a special built-in datatype delivered for assigning XML content as a possible literal value to a target nodes in the RDF graph. In the specification this datatype is described as an instance of rdfs:Datatype and a subclass of rdfs:Literal (using RDFS vocabulary).

rdf:value – is an instance of rdf:Property that may be used in describing structured values. For example, if there is a set of properties describing one subject, one of them can deliver the actual value, while the others – additional information (distance can have rdf:value property with a value, for example "15"^^xsd:decimal, and ex:unit property with a value, for example "meter"^^xsd:string). rdf:value has no meaning on its own. It is provided as a piece of vocabulary that may be used in such idioms. The rdfs:domain and rdfs:range of rdf:value is rdfs:Resource (using RDFS vocabulary).

rdf:Statement – is a resource reifying a triple. Such a resource must have at least 3 properties: rdf:subject, rdf:object and rdf:predicate, valued by the corresponding resources.

rdf:Alt, rdf:Bag, rdf:Seq – these concepts are used in the description of containers. rdf:Bag represents a container of unordered elements with duplicates allowed. rdf:Alt is a container of alternative elements, possibly with a preference ordering, from which one is to be selected. rdf:Seq is a container of ordered elements. All three concepts characterize the types of containers and provide the information on partial enumeration of their items rather then construct these containers. All they use the rdf:_n to establish the containment relationship with other resources.

rdf:_1, rdf_2, ... – these are blank nodes of an RDF graph, which are not absolutely identified by URIs. They represent anonymous resources at which RDF graph branches. They are properties that associate a container as the subject with a resource it contains as the object.

rdf:List – is an instance of rdfs:Class that can be used to create collections known as list or list-like structures. Declaration of such collections is similar as in the programming languages, with head, and tail, and terminator declarations (for which the concepts of rdf:first, rdf:last, rdf:nil are used, respectively).

rdf:first – this concept is used in the description of list and other list-like structures. It appears in the triples of the form L rdf:first O. The meaning of such triple is following: there is a first-element relationship between L and O. rfd:first is an instance of rfd:Property. The rdfs:domain of rdf:first is rdf:List and its rdfs:range is rdfs:Resource.

rdf:rest – this concept is used in the description of list and other list-like structures. It appears in the triples of the form L rdf:last O. The meaning of such triple is following: there is a rest-of-list relationship between L and O. rfd:last is an instance of rfd:Property. The rdfs:domain of rdf:last is rdf:List and its rdfs:range is rdfs:List.

rdf:nil – is an instance of rdf:List, representing the empty list or list-like structure. rdf:nil appears in the triples of the form L rdf:rest rdf:nil, which, can be read as following: L is an instance of rdf:List that has one item, that can be indicated using rdf:first property.

8.2.2. RDFS vocabulary

The RDFS vocabulary extends the RDF vocabulary. It relates the RDF classes and properties into taxonomies and allows introduction of new taxonomies of classes and properties. New classes can be also defined as specialization of old ones.

The RDFS concepts have URI reference with leading substring `http://www.w3.org/2000/01/rdf-schema#` associated, by convention, with a namespace prefix `rdfs:`.

In RDFS a class may be an instance of a class. All resources are instances of the class `rdfs:Resource`. All classes are instances of `rdfs:Class` and subclasses of `rdfs:Resource`. All literals are instances of `rdfs:Literal`. All properties are instances of `rdf:Property`.

The RDFS specification provides mechanism for describing limitations on the acceptable types of values assigned to the properties or on the classes to which such properties can be sensibly ascribed. The definitions of constrains can be done on the domain and range of RDF properties. Domain of a property states that any resource that has given property is an instance of the class. Range of a property states that the values of a property are instances of the class. If multiple classes are defined as the domain and range then the intersection of these classes is used. This allows inference of implicit triples (by the inference engines) and schema validation. The list of properties defined in RDF and RDFS vocabularies, together with their domain and range specification, is shown in Table 8.4.

Elements of RDFS vocabulary can be used in recursive declarations (as, for example, a concept `rdfs:range` is used to declare the range of `rdfs:range`). RDFS includes concepts allowing creation of higher-level languages on it. RDFS vocabulary helps in specification of the information; it does not indicate how this information should be used.

Table 8.2. RDFS vocabulary.

rdfs:domain	rdfs:range	rdfs:Resource	rdfs:Literal
rdfs:Datatype	rdfs:Class	rdfs:subClassOf	rdfs:subPropertyOf
rdfs:member	rdfs:Container	rdfs:ContainerMembershipProperty	
rdfs:comment	rdfs:seeAlso	rdfs:isDefinedBy	rdfs:label

`rdfs:domain` – is an instance of `rdf:Property` used in resource definitions. These definitions states that any resource with `rdfs:domain` property given is an instance of one or more classes. Thus, the use of `rdfs:domain` is similar to the type declaration. The usage of `rdfs:domain` is following: `P rdfs:domain C`. The triples declared in that way have the following interpretation: `P` is an instance of the class `rdf:Property`, and `C` is an instance of the class `rdfs:Class`, and the resources denoted by the subjects of triples whose predicate is `P` are instances of the class `C`.

The property `P` can have more than one `rdfs:domain` property assigned. As a consequence, the resources denoted by subjects of any triples with predicate `P` are instances of all the classes stated by the `rdfs:domain` properties (are instances of more then one class).

The domain of `rdfs:domain` is `rdf:Property` class. It means that domain property applies to instances, which are properties themselves. The range of `rdfs:domain` is `rdfs:Class` class. This states that any resource that is the value of an `rdfs:domain` property is an instance of `rdfs:Class`.

`rdfs:range` – is an instance of `rdf:Property` that is used for range of property values definition. `rdfs:range` states that values of a property are instances of one or more classes. Thus, the use of `rdfs:range` is similar to the type declaration. `rdfs:range` appears in the triples in form: `P rdfs:range C`. The interpretation of such triple is following: `P` is an instance of the class `rdf:Property`, `C` is an instance of the class `rdfs:Class` and the resources denoted by the objects of triples whose predicate is `P` are instances of the class `C`. The property `P` can have more than one `rdfs:range` property assigned. In consequence, the resources denoted by the objects of triples with predicate `P` are instances of all the classes stated by the `rdfs:range` properties.

The domain of `rdfs:range` is `rdf:Property` class. Thus any resource with an `rdfs:range` property is an instance of `rdf:Property`. The range of `rdfs:range` is the `rdfs:Class`. This states that any resource that is the value of an `rdfs:range` property is an instance of `rdfs:Class`.

`rdfs:Resource` – is a class, whose instances can be all things described by RDF as resources. Thus, it is the class of everything. All other classes are subclasses of this class. `rdfs:Resource` is an instance of `rdfs:Class`.

`rdfs:Literal` – is a class of literals representing values such as numbers and dates. Its intended use is to be the range of properties. Anything represented by a literal could also be represented by a URI, but it is often more convenient or intuitive to use literals. Literals may be plain (string combined with an optional language tags, as `"October, 1, 2010"`) or typed (string combined with a datatype URI, as `"2010-01-10"^^xsd:date`). In both cases the string is recommended to be in Unicode Normal Form C (defined in the document available at: http://www.unicode.org/unicode/reports/tr15/). Typed `rdfs:Literal` is an instance of `rdfs:Class`. `rdfs:Literal` is a subclass of `rdfs:Resource`.

`rdfs:Datatype` – is a class of datatypes. A datatype is identified by one or more URI references. Each instance of `rdfs:Datatype` is a subclass of `rdfs:Literal`. `rdfs:Datatype` is both an instance of and a subclass of `rdfs:Class`.

`rdfs:Class` – is a concept used in RDF class. Classes are themselves resources. Once declared, the RDF class can be used as a value of `rdf:type` property. The subject of the corresponding triple becomes implicitly an instance of the class. The members of a RDF class are instances of the class. However, the class and a set of class' instances do not have to be the same. The set of instances is the extension of the class, and two different classes may contain the same set of instances. `rdfs:Class` is an instance of `rdfs:Class`, and is the class of classes. The group of resources that are RDF Schema classes is itself a class called `rdfs:Class`.

`rdfs:subClassOf` – is a property used to form a taxonomy of classes by extending existing classes. It might be used to state that one class is a subclass of another. Extension reuses (and thus shares) existing definition(s). A class can have multiple superclasses. If a class `C` is a subclass of a class `C'`, then all instances of `C` will also be instances of `C'`. The term super-class is used as the inverse of subclass. If a class `C'` is a super-class of a class `C`, then all instances of `C` are also instances of `C'`.

`rdfs:subPropertyOf` – is a property used to form a taxonomy of properties in a similar way as `rdfs:subClassOf` in a classes case.

`rdfs:member` – is a property that is a super-property of all the container membership properties, each container membership property has an `rdfs:subPropertyOf` relationship to the property `rdfs:member`.

rdfs:Container – is a class used to represent the core RDF Container classes, i.e. rdf:Bag, rdf:Seq, rdf:Alt.

rdfs:ContainerMembershipProperty – is a class, instances of which are properties: rdf:_1, rdf:_2, rdf:_3 ... stating that a resource is a member of a container. rdfs:ContainerMembershipProperty is a subclass of rdf:Property. Each instance of rdfs:ContainerMembershipProperty is an rdfs:subPropertyOf the rdfs:member property. Container membership properties might be applied to resources other than containers.

rdfs:comment – is a an instance of rdf:Property that may be used to provide a human-readable description of a resource, clarifying its meaning. Multilingual documentation is supported through use of the language tagging facility of RDF literals.

rdfs:seeAlso – is an instance of rdf:Property that is used to indicate a resource that might provide additional information about the subject resource.

rdfs:isDefinedBy – is an instance of rdf:Property that is used to indicate a resource defining the subject resource. This property might be used to indicate an RDF vocabulary in which a resource is described. rdfs:isDefinedBy is a subproperty of rdfs:seeAlso.

rdfs:label – is an instance of rdf:Property that may be used to provide a human-readable version of a resource's name. Multilingual labels are supported using the language tagging facility of RDF literals.

RDF and RDFS concepts are related. All RDF and RDFS classes with corresponding subclass-of (rdfs:subClassOf) and instance-of (rdf:type) relations are shown in Table 8.3. The domains and ranges of build-in RDF and RDFS properties (instances of rdf:Property) are listed in Table 8.4.

Table 8.3. Relations between RDF and RDFS concepts.

Element	Class of	rdfs:subClassOf	rdf:type
rdfs:Resource	all resources	rdfs:Resource	rdfs:Class
rdfs:Class	all classes	rdfs:Resource	rdfs:Class
rdfs:Literal	literal values	rdfs:Resource	rdfs:Class
rdfs:Datatype	datatypes	rdfs:Class	rdfs:Class
rdf:XMLLiteral	XML literal values	rdfs:Literal	rdfs:Datatype
rdf:Property	properties	rdfs:Resource	rdfs:Class
rdf:Statement	statements	rdfs:Resource	rdfs:Class
rdf:List	lists	rdfs:Resource	rdfs:Class
rdfs:Container	containers	rdfs:Resource	rdfs:Class
rdf:Bag	unordered containers	rdfs:Container	rdfs:Class
rdf:Seq	ordered containers	rdfs:Container	rdfs:Class
rdf:Alt	containers of alternatives	rdfs:Container	rdfs:Class
rdfs:Container MembershipProperty	rdf:_1... properties expressing membership	rdf:Property	rdfs:Class

Table 8.4. Roles and restrictions of RDF and RDFS properties.

Element	Role	rdfs:domain	rdfs:range
rdfs:range	restriction on subject	rdf:Property	rdfs:Class
rdfs:domain	restriction on object	rdf:Property	rdfs:Class
rdf:type	instance declaration	rdfs:Resource	rdfs:Class
rdfs:subClassOf	subclass declaration	rdfs:Class	rdfs:Class

rdfs:subPropertyOf	subproperty declaration	rdf:Property	rdf:Property
rdfs:label	human readable label	rdfs:Resource	rdfs:Literal
rdfs:comment	human readable comment	rdfs:Resource	rdfs:Literal
rdfs:member	container membership	rdfs:Resource	rdfs:Resource
rdf:first	first element declaration	rdf:List	rdfs:Resource
rdf:rest	rest of a list declaration	rdf:List	rdf:List
rdf:_1,rdf:_2, ...	container membership	rdfs:Container	rdfs:Resource
rdfs:seeAlso	additional information	rdfs:Resource	rdfs:Resource
rdfs:isDefinedBy	subject definition info	rdfs:Resource	rdfs:Resource
rdf:value	used for structured values	rdfs:Resource	rdfs:Resource
rdf:object	object declaration	rdf:Statement	rdfs:Resource
rdf:predicate	predicate declaration	rdf:Statement	rdfs:Resource
rdf:subject	subject declaration	rdf:Statement	rdfs:Resource

8.3. RDF serialization

A natural representation of RDF graphs is graphical representation, with nodes and arcs reflecting particular triples. This representation is good for human analysis but not for information exchange or processing by computers. In order to handle RDF graphs by computers the graphs have to be serialized.

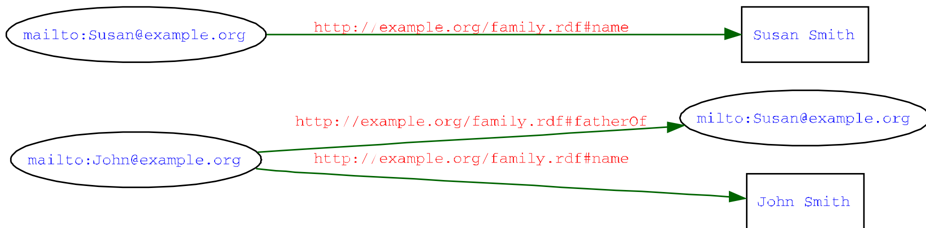
Serialization provides a way to convert between the abstract RDF model to a concrete format, such as a file or other byte stream. The most popular methods of RDF graphs serialization are: RDF/XML, Terse RDF Triple Language (Turtle), and N-Triples. The serialization preserves all the constructs of the original RDF graph. Hence, serialized graphs carry the same information, independently of the serialization method used. The only difference is that, in general, this information is formatted more or less conveniently using serialization method specific features.

The description of RDF/XML syntax with explanation about RDF graphs encoding can be found on the Web at `http://www.w3.org/TR/rdf-syntax-grammar/` . More information about Turtle can be found on the Web at `http://www.w3.org/TeamSubmission/turtle/` . More details on N-Triples can be found on the Web at `http://www.w3.org/TR/rdf-testcases/#ntriples` .

NOTE:

One of the key properties of RDF graphs is that they do not have roots. No single resource is of any inherent significance as compared to other. Thanks to this, combining RDF graphs is conceptually the same as placing them next to one another.

There are some very useful RDF tools available on-line allowing conversions between different serialization formats and visualization of RDF graphs. A W3C validator for RDF/XML (available at http://www.w3.org/RDF/Validator/) can validate and visualize RDF/XML input. Another validator (available at http://www.rdfabout.com/demo/validator/validate.xpd) can validate input, which can be a serialized RDF graph in one of the mentioned representations, and produce output in other representation. The example of a visualized RDF graph together with its all three serialized forms is presented in Figure 8.4. The W3C validator for XML/RDF input has generated the diagram. Please note, that this is the same graph as in Figure 8.3, but with the triples separated.



RDF/XML serialization

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
xmlns:ex="http://example.org/family.rdf#">
      <rdf:Description rdf:about="mailto:John@example.org">
            <ex:fatherOf rdf:resource="milto:Susan@example.org" />
            <ex:name
rdf:datatype="http://www.w3.org/2001/XMLSchema#string">John Smith</ex:name>
      </rdf:Description>
      <rdf:Description rdf:about="mailto:Susan@example.org">
            <ex:name
rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Susan
Smith</ex:name>
      </rdf:Description>
</rdf:RDF>
```

TURTLE serialization

```
@prefix ex: <http://example.org/family.rdf#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
# This is a comment.
<mailto:John@example.org>
ex:fatherOf <milto:Susan@example.org> ;
ex:name "John Smith"^^xsd:string .
<mailto:Susan@example.org> ex:name "Susan Smith"^^xsd:string .
```

```
N-Triples serialization
<mailto:John@example.org>          <http://example.org/family.rdf#fatherOf>
<milto:Susan@example.org> .
<mailto:John@example.org>    <http://example.org/family.rdf#name>    "John
Smith"^^<http://www.w3.org/2001/XMLSchema#string> .
<mailto:Susan@example.org>    <http://example.org/family.rdf#name>    "Susan
Smith"^^<http://www.w3.org/2001/XMLSchema#string> .
```

Figure 8.4. The example of an RDF graph serialization (visualized on top by the W3C validator for XML/RDF).

8.3.1. RDF/XML

RDF/XML is a normative, platform independent XML syntax for representing RDF triples. Thanks to RDF/XML the RDF graphs can be serialized and exchanged between different computers running different operating systems. Unfortunately, there is no canonical RDF/XML serialization procedure. Thus, the results of serialization are not necessarily unique, i.e. the same RDF graph can be represented in RDF/XML in different ways. This introduces some difficulties when comparing content of two documents – the same content for the same model is not guaranteed. For example, two following RDF/XML serializations are equivalent:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:ex="http://example.org/terms#" >
<ex:Description rdf:about="http://example.org/myFamily.rdf#John">
  <rdf:type rdf:resource="http://example.org/terms#Man" />
  <ex:fatherOf rdf:resource=http://example.org/myFamily.rdf#Susan />
  <ex:name>John Smith</ex:name>
</ex:Description>
<ex:Woman rdf:about="http://example.org/myFamily.rdf#Susan">
  <ex:name>Susan Smith</ex:name>
</ex:Woman>
</rdf:RDF>
```

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:ex="http://example.org/terms#" >
<ex:Man rdf:about="http://example.org/myFamily.rdf#John"
 ex:name="John Smith">
  <ex:fatherOf>
    <ex:Woman rdf:about="http://example.org/myFamily.rdf#Susan"
     ex:name="Susan Smith" />
   </ex:fatherOf>
  <ex:name>John Smith</ex:name>
</ex:Man >
</rdf:RDF>
```

Any RDF/XML document starts with the `rdf:RDF` element declaration, within which the whole serialized RDF graph is embedded as a series of `rdf:Description` elements. The list of attributes of `rdf:RDF` element contains XML namespace declarations. By convention of the Semantic Web community, the namespace associated with `rdf` prefix is `http://www.w3.org/1999/02/22-rdf-syntax-ns#`. This is always true, either for XML namespaces declared in RDF/XML or by similar facilities offered by other serialization methods. The definitions of elements used in RDF/XML serialization are given in RDF/XML document, which is available under the link `http://www.w3.org/1999/02/22-rdf-syntax-ns`. That document defines RDF itself (RDF Schema for the RDF vocabulary defined in the RDF namespace).

Statements: In general RDF/XML statements are declared in the scope of `<rdf:Description>` elements. One `<rdf:Description>` element can group several statements with the same subject and different predicates and objects. The subject is declared as a value of `rdf:about` element's attribute. Names of nested elements or names of element's atrributes represent predicates. If an object of the statement is a resource, it is represented by the value of `rdf:resource` predicate's attribute. If an object of the statement is a literal, it is represented by the predicate's content (if the predicate is an element) or by the predicate's value (if the predicate is an attribute).

In the example below the subject identified as a resource `http://example.org/myFamily.rdf#John` has `rdf:type` property and the value of this property, i.e. object, is `http://example.org/terms#Father`.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
 <rdf:Description rdf:about="http://example.org/myFamily.rdf#John">
   <rdf:type rdf:resource="http://example.org/terms#Father" />
 </rdf:Description>
</rdf:RDF>
```

The use of `rdf:type` predicate in this way is very common for RDF/XML. The node elements declared in such a way are called *typed node elements*. There is a shorthand syntax

for expressing typed node elements: `<`*type* `rdf:about="`*resource*`" />`. Thus, the `rdf:Description` tag is replaced with the namespaced-element corresponding to the RDF URI reference of the value of the `rdf:type` predicate and `rdf:type` predicate is omitted (in a case of multiple `rdf:type` predicates only one can be used this way, the others must remain as property elements or property attributes).

RDF/XML allows a further abbreviation. This incorporates the use of `xml:base` attribute for setting the base URI for resolving relative RDF URI references (otherwise the base URI of the document is valid). The base URI set applies to all RDF/XML attributes that deal with RDF URI references which are `rdf:about`, `rdf:resource`, `rdf:ID` and `rdf:datatype`. Additionally, the `rdf:ID` attribute on a node element (not a property element, that has another meaning) can be used instead of `rdf:about`. When used, it gives a relative RDF URI reference equivalent to `#` concatenated with the `rdf:ID` attribute value. `rdf:ID` is useful for defining a set of distinct, related terms relative to the same RDF URI reference. Such terms cannot appear more than once in the scope of an `xml:base` value (or document, if none is given) what is automatically checked by XML editing tools.

In the example below, there are two alternative declarations of the same statement already presented.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xml:base="http://example.org/myFamily.rdf"
    xmlns:ex="http://example.org/terms#" >
  <ex:Father rdf:about="#John" />
  <ex:Father rdf:ID="John" />
 </rdf:RDF>
```

Resources: In RDF/XML resources can appear as elements' names (subjects or predicates) or values of attributes (objects). In the former case, the URI identifying a resource has to be abbreviated using standard XML namespace conventions (like `ex:Father`). In the latter case, the URI can be abbreviated applying XML entity declarations (like `rdf:about="&base;#John"`). The example below shows both cases.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE rdf:RDF [
<!ENTITY base "http://example.org/myFamily.rdf">
]>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xml:base="&base;"
    xmlns:ex="http://example.org/terms#" >
 <ex:Father rdf:about="&base;#John" />
 <ex:Father rdf:ID="John" />
</rdf:RDF>
```

Literals: In RDF/XML literals can appear in an every place where a property value is expected: as attributes of elements or as contents of property nodes but with some restrictions. The plain literals can appear as either property attributes or property nodes values, while the typed literals (XML or custom typed) only as property node contents. In the example below the literal `"John Smith"` appeared as a value of `ex:name` property attribute (declared in the namespace `xmlns:ex="http://example.org/terms#"`).

```
<ex:Father rdf:about="#John" ex:name="John Smith"/>
```

The same effect can be received by inserting this literal into the content of a property node `<ex:name>` as in the example below:

```
<rdf:Description rdf:about="http://example.org/myFamily.rdf#John">
    <rdf:type rdf:resource="http://example.org/terms#Father" />
      <ex:name>John Smith</ex:name>
 </rdf:Description>
```

Plain literals, when declared as a property node content, can have an optional indicator of the content's language. This indicator is provided as a value of an optional `rdf:lang` attribute. In fact, this attribute can be used on any node element or property element to indicate that the included content is in the given language. The set of valid language indicators is restricted. All language indicators must be lowercased and match language tags as defined by RFC 4646 (available at http://www.ietf.org/rfc/rfc4646.txt).

For an XML literal, an attribute `rdf:parseType` should be used with a value set to `"Literal"` string. If so, the contents of the property node can be any XML document (as shown in the example below).

```
<rdf:Description rdf:about="http://example.org/myFamily.rdf#John's_car">
    <ex:record rdf:parseType="Literal" >
        <ex:color>red</ex:color>
         <ex:checkDates>
                  <ex:lastCheck>2009-02-03</ex:lastCheck>
                  <ex:nextCheck>2012-02-02</ex:nextCheck>
         </ex:checkDates>
        </ex:record>
 </rdf:Description>
</rdf:RDF>
```

Typed literals have the type of the content declared using `rdf:datatype` attribute. The value of this attribute should be a datatype URI as defined in XML Schema or a custom datatype URI. In the example below `"33"` string will be interpreted as an integer number.

```
<ex:age rdf:datatype="http://www.w3.org/2001/XMLSchema#int">33</ex:age>
```

Comments: As in any other XML document comments can be provided within tags composed from characters `<!--` and `-->`. But comments are not part of RDF graph and can disappear in serializing-deserialising round-trip.

Blank nodes: In RDF-XML a b-node can be represented by a nested element. The example below illustrates how to use a b-node when serializing the sentence like this: "John likes somebody who is a woman and has name Ann" (see Figure 8.5).

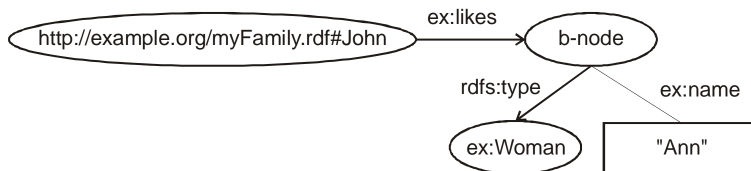


Figure 8.5. Example of the graph with a blank node.

The blank node represents an unreferenced person – there is no URI identifying this individual. We only know that the type of the person is an `ex:Woman` and that her name is `Ann`. The RDF/XML serialization of the graph discussed is as follows:

```
<Description rdf:about="http://example.org/myFamily.rdf#John">
 <ex:likes>
<ex:Woman ex:name="Ann" />
 </ex:likes>
</Description>
```

The same blank node can be used more that once in the same RDF graph. To allow such a multiple use an attribute `rdf:nodeID` is used.

The attribute assignment `rdf:nodeID="b-node identifier"` replaces `rdf:about="RDF URI reference"` when declaring blank node (the place for declaration of b-node with an identifier is `Description` element) or replaces `rdf:resource="RDF URI reference"` when declaring property element (the place where reference to b-node identified is used is nested element).

In the example below, the sentence “John and Adam like someone whose name is Meryl Streep” is represented in RDF/XML syntax using a blank node.

```
<Description rdf:about="http://example.org/myFamily.rdf#John">
 <ex:likes rdf:nodeID="b1"/>
</Description>
<Description rdf:about="http://example.org/myFamily.rdf#Adam">
 <ex:likes rdf:nodeID="b1"/>
</Description>
<Description rdf:nodeID="b1">
      <ex:name>Meryl Streep</ex:name>
</Description>
```

Blank nodes can be declared omitting the `<rdf:Description>` `</rdf:Description>` pair. This requires insertion of `rdf:parseType="Resource"` attribute on the containing property element. Such insertion turns the property element into a property-and-node element, which can itself have both property elements and property attributes (but without `rdf:nodeID` attribute at the same time).

```
<Description rdf:about="http://example.org/myFamily.rdf#John">
 <ex:likes rdf:parseType="Resource">
<ex:name>Meryl Streep</ex:name>
<ex:profession>actress</ex:profession>
 </ex:likes>
</Description>
```

There is a way for further simplification. If all of the property elements on a blank node element have string literal values with the same in-scope `xml:lang` value (if present) and each of these property elements appears at most once and there is at most one `rdf:type` property element with a RDF URI reference object node, these elements can be omitted. But this abbreviation requires transforming omitted elements into property attributes on the containing property element, which becomes then an empty element.

```
<Description rdf:about="http://example.org/myFamily.rdf#John">
 <ex:likes ex:name="Meril Strip" ex:profession="actress"/>
</Description>
```

Containers and collections: In RDF/XML serialization containers are declared as nested `rdf:Bag`, `rdf:Seq`, `rdf:Alt` elements – nodes of the RDF graph with a type property reflecting container's type. The `rdf:about` attribute can provide URIs identifying these nodes. Without this attribute, any given container becomes b-node. The elements nested in a container are `rdf:_n` or `rdf:li` . These elements are interpreted as properties of the container's node with values defined by `rdf:resource` attribute. The example of all three types of containers declaration is provided below.

```
<?xml version="1.0"?>
<rdf:RDF                                      xmlns:ex="http://example.org/"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
      <rdf:Description rdf:about="http://example.org/Tournament">
            <ex:hasCompetitors>
              <rdf:Bag rdf:about="http://example.org/Competitors">
                  <rdf:li rdf:resource="http://example.org/Adam" />
                  <rdf:li rdf:resource="http://example.org/Witold" />
                  <rdf:li rdf:resource="http://example.org/Tomasz" />
              </rdf:Bag>
            </ex:hasCompetitors>
            <ex:hasStages>
              <rdf:Seq rdf:about="http://example.org/Stages">
                  <rdf:li rdf:resource="http://example.org/Preliminary" />
                  <rdf:li rdf:resource="http://example.org/Group" />
                  <rdf:li rdf:resource="http://example.org/Final" />
              </rdf:Seq>
            </ex:hasStages>
            <ex:hasPlace>
              <rdf:Alt rdf:about="http://example.org/Playgrounds">
                  <rdf:li rdf:resource="http://example.net/Playground1" />
                  <rdf:li rdf:resource="http://example.net/Playground2" />
              </rdf:Alt>
            </ex:hasPlace>
      </rdf:Description>
</rdf:RDF>
```

8.3.2. Terse RDF Triple Language (Turtle)

The Terse RDF Triple Language (Turtle) is the simplest and most concise serialization syntax for RDF used in many textbooks and tutorials. Its human-friendly and readable syntax was designed specifically for RDF. Turtle is not an XML language, and therefore it has no support from XML editors.

Statements: The Turtle syntax of writing RDF statements is quite simple. All parts of the statement, as subject, predicate, and object, should be written in one line, separated by white spaces, and terminated with a period. The statements can be written in consecutive lines or, if there are multiple statements about the same subject, they can be written in a shorthand way. This shorthand way relies on writing shared subject followed by a sequence of pairs composed from predicate and object of the statements, separated with a semicolon and terminated with a period. In the example there are two statement declared. `exf:John` is a

subject of both statements, `rdf:type` and `ext:name` are properties, `exf:Father` and `"John Smith"` are objects (the prefixes declarations were omitted).

```
exf:John rdf:type ext:Father .
exf:John ext:name "John Smith" .
```

An equivalent, shortened form of these declarations is as follows:

```
exf:John rdf:type ext:Father ;
ext:name "John Smith" .
```

Similar shorthand can be applied when shortening statements having the same both subject and predicate. In such cases, the shared subject and predicate should be followed by the objects of statements, separated with a comma and terminated with a period. Thus the shortened form of two statements declaration:

```
exf:John ext:likes "Meryl Streep" .
exf:John ext:likes "Another name".
```

can be written as follows:

```
exf:John ext:likes "Meryl Streep", "Another name" .
```

The reification of statements can be written in a similar manner as presented below:

```
[ a rdf:Statement;
rdf:subject exf:John;
rdf:predicate ext:likes;
rdf:object "Meryl Streep" ] .
```

Resources: The use of URIs in Turtle for RDF resources identification has something in common with the declarations and uses of XML namespaces. URI can be defined as either fully qualified identifier or identifier built from a declared prefix and extension. In the former case, the URI appearing in the statements definitions should be enclosed within angled brackets: < and >. In the later case, the declaration of the prefix should be written in a line, starting with `@prefix` keyword, followed by the prefix name and a leading part of URIref enclosed within angled brackets. All these three parts should be separated by white spaces. After a declaration, the prefix can be used in any statement definition.

The type of the resource can be declared with `rdf:type` predicate, written on line between the resource and the type URI. Designating the type of an individual resource is also possible with the use of convenient shorthand. The syntax of it is similar to the use of `rdf:type` predicate, with the character `a` used instead of `rdf:type`. In the example below, there is one statement about `John` (subject), whose `type` (predicate) is `Father` (object). The example starts with the declarations of prefixes used.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix ext: <http://example.org/terms#> .
@prefix exf: <http://example.org/myFamily.rdf#> .
# the statement example comment
ex:John rdf:type ex:Father .
```

The equivalent statement using a syntactic shorthand for `rdf:type` is following:

```
ex:John a ex:Father .
```

Using fully qualified URIs in the presented example results in a statement of the form:

```
<http://example.org/terms#John>    <http://www.w3.org/1999/02/22-rdf-syntax-
ns#type> <http://example.org/terms#Father> .
```

Literals: Writing literals in Turtle depends on whether they are plain, data-typed or language tagged, as in RDF/XML. In general, literals are written as strings enclosed in double quotes (in a case of a string without the line break character) or as strings limited by the set of three double quotes on both sides (in a case of string containing line break). Since double quote is a special character, it appears in the literal-string values written as \" (U+0022). Similar escapes are used to encode surrounding syntax, non-printable characters and to encode Unicode characters by codepoint number (although they may also be given directly, encoded as UTF-8). These escapes are: `\t` (U+0009, tab), `\n` (U+000A, linefeed), `\r` (U+000D, carriage return), `\>` (U+003E, greater than - only allowed inside URIs), `\\` (U+005C, backslash), `\uHHHH` or `\UHHHHHHHH` (Unicode characters defined by hexadecimal codepoint, where `H` is a single hexadecimal digit).

```
@prefix ex: <http://example.org/> .
ex:Book ex:hasMotto """Here is the first line of the  \"motto\",
and next line,
and final line.""" .
```

The data typed literals are written with `^^` suffix, followed by any legal URI form giving the datatype URI. The language tagged literals are written with `@` suffix followed by the valid character language tag. Literals might be given either a language suffix or a datatype URI but never both.

```
@prefix ex: <http://example.org/> .
ex:Bridge ex:numberOfCards "52"^^<http://www.w3.org/2001/XMLSchema#int> ;
ex:Bridge ex:name "Bridge"@en ;
ex:Bridge ex:name "Brydż"@pl.
```

Comments: Lines starting with `#` character are comments. The Turtle parser will ignore all text after this character to the end of the line.

Blank nodes: The syntax of Turtle reserved a special notion of a b-node identifier. Such identifier starts with a prefix, which is a colon, followed by a node ID. For example `:b3` is a valid b-node identifier. Each b-node identifier is unique only within the scope of a single RDF document. There is also a shorthand notion used when referring to the blank nodes. It is possible to define a blank node without b-node identifier. This can be done with the use of a pair of square brackets `[` and `]`. All statements written within these brackets have an unnamed b-node as the subject.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
ex:Furniture ex:hasDescription :Furniture-01 .
:Furniture-01
ex:name "chair"@en, "krzesło"@pl;
```

```
ex:color "brown"@en, "brązowy"@pl;
ex:productionDate "2010-12-01"^^<http://www.w3.org/2001/XMLSchema#date> .

@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
ex:Furniture ex:hasDescription [ex:name "chair"@en, "krzesło"@pl;
ex:color "brown"@en, "brązowy"@pl;
ex:productionDate "2010-12-01"^^<http://www.w3.org/2001/XMLSchema#date>] .
```

In the following document, both lines after the prefix declarations part are equivalent:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix ex: <http://example.org/terms#>
@prefix : < http://example.org/myFamily.rdf > .
:#John a ex:Father .
: #John rdf:type ex:Father .
```

Containers and collections: A container declaration in Turtle starts with a line containing the subject, keyword `a` and one of the following terms: `rdf:Bag`, `rdf:Seq` and `rdf:Alt`. The consecutive line includes a list of predicates of the form, `rdf: 1`, `rdf: 2`, `rdf: 3`, . . . , `rdf n` together with associated resources. All lines should end with a semicolon, except last, ended by dot. The numbers near by `rdf:` terms can be ignored in the declarations of `rdf:Bag` and `rdf:Alt` containers, but not in `rdf:Sequence`. Ordering of elements in `rdf:Sequence` is significant. If the sequence is declared once, using predicates as `rdf: n` is straightforward. However, inserting any new elements into an existing structure can cause some problems. If this did happen, several predicates would need to be re-enumerated. The solution to this problem is given with the use of the `rdf:li` predicate. This predicates substitutes any of `rdf: n` predicates. When used, the order in which `rdf:li` predicates appear in the document is significant. The first resource of the group associated with `rdf:li` becomes `rdf: 1`, the second `rdf: 2`, and so on. Resources declared in such a way will not be altered even when different RDF graphs are merged.

```
@prefix ex: <http://example.org/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
ex:Competitors a rdf:Bag ;
rdf:_1 ex:Adam ;
rdf:_2 ex:Witold ;
rdf:_3 ex:Tomasz .
ex:Stages a rdf:Seq ;
rdf:_1 ex:Preliminary ;
rdf:_2 ex:Group ;
rdf:_3 ex:Final .
ex:Playgrounds a rdf:Alt ;
rdf:_1 <http://example.net/Playground1> ;
rdf:_2 <http://example.net/Playground2> .
ex:Tournament ex:hasCompetitors ex:Competitors .
ex:Tournament ex:hasStages ex:Stages .
ex:Tournament ex:hasPlace ex:Playgrounds .
```

The collection declaration in Turtle is based on the concept of head-tail links. This concept is quite similar to the concept of the list data structure. The declaration starts with a subject, keyword `a`, followed by the `rdf:List` term. Two next lines starts with, respectively, `rdf:first` (head) and `rdf:rest` (tail) predicates. A resourse identifier follows each predicate. `rdf:first` predicate usually refers to the object, while `rdf:rest` refers to the

b-node being a subjects of another collection declaration. Thus, the collection declaration includes elements that explicitly refer to the subsequent elements. The objects of `rdf:first` predicates are members of collection being declared. The terminator of such recursive declaration is `rdf:rest` predicate, whose object is `rdf:nil` (the tail referring to nil).

```
@prefix ex: <http://example.org/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix cmp: <http://example.org/Company#> .
ex:Sponsors a rdf:List ;
rdf:first cmp:Company1 ;
rdf:rest :r1 .
:r1 a rdf:List ;
rdf:first cmp:Compan2 ;
rdf:rest :r2 .
:r2 a rdf:List ;
rdf:first cmp:Company3 ;
rdf:rest :r3 .
:r3 a rdf:List ;
rdf:first ex:Company4;
rdf:rest rdf:nil .

ex:Tournament ex:hasSponsors ex:Sponsors .
```

There is also concise shorthand for representing collections. The lists can be written as a collection of white space-separated resources enclosed within parenthesis.

```
ex:Tournament ex:hasSponsors (cmp:Company1 cmp:Company2 cmp:Company3) .
```

8.3.3. N-Triples

Serialization with N-Triples is based on the same syntax for comments, resources and literal values as in Turtle, but imposes some restrictions. These restrictions obey missing `@prefix` directive, missing shorthand notion with semicolon or coma, and necessity of writing statements (triples) in a single line.

8.4. RDF Applications

8.4.1. Dublin Core, FOAF

Many languages introduce new vocabulary terms based on RDF. Two of them, FOAF (Friend of a Friend, `http://www.foaf-project.org/`, and DC (Dublin core, `http://dublincore.org/`) are widely used as supporting vocabularies in semantic modelling. Both can be used in a definition of custom ontology (see the example below).

```
RDF/XML serialization
<rdf:RDF
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:foaf="http://xmlns.com/foaf/0.1/"
     xmlns="http://www.example.org/contact.rdf#">
   <foaf:Person rdf:about="http://www.example.org/contact.rdf#johnsmith">
   <foaf:mbox rdf:resource="mailto:john.smith@example.org"/>
   <foaf:homepage rdf:resource="http://www.example.org/~jsmith/"/>
   <foaf:family_name>Smith</foaf:family_name>
```

```
   <foaf:givenname>John</foaf:givenname>
   </foaf:Person>
</rdf:RDF>
```

```
Turtle serialization
@prefix :     <http://www.example.org/contact.rdf#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

:johnsmith a foaf:Person ;
    foaf:givenname "John" ;
    foaf:family_name "Smith" ;
    foaf:homepage <http://www.example.org/~jsmith/> ;
    foaf:mbox <mailto:joe.smith@example.org> .
```

8.4.2. RDF API

JENA

Jena is a Java based open source framework for building Semantic Web applications (http://jena.sourceforge.net/), providing RDF and OWL APIs. It allows reading and writing RDF in various formats (RDF/XML, N3 and N-Triples). It offers in-memory and persistent storage, supports SPARQL query engine and includes a rule-based inference engine. The example presented below shows how to generate an RDF file with the use of the Jena framework.

```
import java.io.*;
import com.hp.hpl.jena.rdf.model.*;
import com.hp.hpl.jena.vocabulary.*;

public class GenerateRDF extends Object {
 public static void main (String args[]) {
    String personURI    = "http://localhost/jkowalski";
    String givenName    = "John";
    String familyName   = "Kowalski";
    String fullName     = givenName+familyName;

    Model model = ModelFactory.createDefaultModel();

    Resource node = model.createResource(personURI)
         .addProperty(VCARD.FN, fullName)
         .addProperty(VCARD.N,
                model.createResource()
                    .addProperty(VCARD.Given, givenName)
                    .addProperty(VCARD.Family, familyName));
    model.write(System.out);
 }
}
```

There are two imported packages in the example: com.hp.hpl.jena.rdf.model and com.hp.hpl.jena.vocabulary. They are necessary for, respectively, creating and manipulating model resources, and creating VCARD vocabulary. The line with the assignment `Model model = ModelFactory.createDefaultModel();` creates an object of Model interface. The next line creates the resource and adds properties to it by using the addProperty() method. After compilation and execution with external jars (jena.jar, icu4j_3_4.jar, iri.jar, commons-logging-1.1.1.jar, xercesImpl.jar) the example produce the following output:

```
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:vcard="http://www.w3.org/2001/vcard-rdf/3.0#" >
  <rdf:Description rdf:nodeID="A0">
    <vcard:Family>Kowalski</vcard:Family>
   <vcard:Given>John</vcard:Given>
  </rdf:Description>
  <rdf:Description rdf:about="http://localhost/jkowalski">
    <vcard:N rdf:nodeID="A0"/>
   <vcard:FN>JohnKowalski</vcard:FN>
  </rdf:Description>
</rdf:RDF>
```

SWI-Prolog

SWI-Prolog is an open-source environment for logic programming (Prolog stands for Programming in Logic). It is available at `http://www.swi-prolog.org/` and includes the `semweb` library (Prolog library based on foreign-language extensions for storing and manipulating RDF triples). This library depends on the RDF parser library, which in turn depends on the XML parser provided by the `sgml` package. SWI-Prolog can handle quite large sets of triples limited only by available memory. The RDF parser converts an RDF-XML document into the triple notation. The library `library(rdf_write)` creates an RDF/XML document from a list of triples. The example presented below shows how to build a simple RDF graph in Prolog.

```
:- module(subprop,[ subprop/0 ]).
:- use_module('../rdf_db').

rdf_db:ns(test, 'http://www.test.org/').

t1 :- rdf_assert(test:a, rdfs:subPropertyOf, test:r1),
      rdf_assert(test:jan, test:a, literal(jan)).

t2 :- rdf_assert(test:a, rdfs:subPropertyOf, test:r1),
      rdf_assert(test:a, rdfs:subPropertyOf, test:r2),
      rdf_assert(test:jan, test:a, literal(jan)).

t3 :- rdf_assert(test:a, rdfs:subPropertyOf, test:r1),
      rdf_assert(test:a, rdfs:subPropertyOf, test:r2),
      rdf_assert(test:b, rdfs:subPropertyOf, test:r3),
      rdf_assert(test:b, rdfs:subPropertyOf, test:r4),
      rdf_assert(test:c, rdfs:subPropertyOf, test:a),
      rdf_assert(test:c, rdfs:subPropertyOf, test:b),
      rdf_assert(test:jan, test:a, literal(jan)).

subprop :- rdf_reset_db, t3,
      rdf_has(test:jan, test:r1, Name),
      Name == literal(jan).
```

Sesame

Sesame is an open source Java framework with an extensible and configurable storage, and inference and querying support for RDF data (`http://www.openrdf.org/`). The framework supports various RDF file formats, query result formats and query languages. It offers also JBDC-like user API, streamlined system APIs and a RESTful HTTP interface supporting the SPARQL Protocol for RDF (to communicate with configurable storage). The

example below shows how to initialize a non-inferencing main-memory repository using the Sesame framework.

```
import org.openrdf.repository.Repository;
import org.openrdf.repository.sail.SailRepository;
import org.openrdf.sail.memory.MemoryStore;
...
Repository myRepository = new SailRepository(new MemoryStore());
myRepository.initialize();
```

9. OWL (Ontology Web Language)

9.1. Ontology and its languages

Since many years, people are trying to build a knowledge model to represent the meaning of concepts from a domain of interest, and their relationships. The concepts can be anything, ranging from abstract ideas and topics up to concrete things. Ontology is a formal representation of knowledge expressed in specialized modelling language. Thus, ontology describes the meaning of knowledge (semantics). According to wikipedia (`http://en.wikipedia.org/wiki/Ontology`):

Ontology (from the Greek nominative ὤν: being, genitive ὄντος: "of being" (neuter participle of εἶναι: "to be") and -λογία, -logia: science, study, theory) is the philosophical study of the nature of being, existence or reality in general, as well as the basic categories of being and their relations. Traditionally listed as a part of the major branch of philosophy known as metaphysics, ontology deals with questions concerning what entities exist or can be said to exist, and how such entities can be grouped, related within a hierarchy, and subdivided according to similarities and differences.

Ontology can be regarded as a science of what kinds and structures of objects, properties and relations between them are. The ontology helps to understand how different pieces of information relate to each other. Moreover, a knowledge model, if built, is not static. It can change according to the knowledge changes. Ontology also provides the basis for inference and reasoning through mapping to a known logical formalism, with consistency checks, checks for unintended relationships, and automatic classification.

In the computer science ontology means a rigorous and exhaustive organization of some knowledge domain, usually hierarchical, containing all the relevant entities and their relations, suitable for reasoning about these entities within that domain, described in languages that are processable by computers.

NOTE

Categorization is the process in which ideas and objects are recognized, differentiated and understood. Categorization implies that objects are grouped into categories, usually for some specific purpose. Ideally, a category illuminates a relationship between the subjects and objects of knowledge. (http://en.wikipedia.org/wiki/Categorization)

Hierarchy (Greek: hierarchia (ἱεραρχία), from hierarches, "leader of sacred rites") is an arrangement of items (objects, names, values, categories, etc.) in which the items are represented as being "above," "below," or "at the same level as" one another. Abstractly, a hierarchy is simply an ordered set or an acyclic graph. (http://en.wikipedia.org/wiki/Hierarchy)

The ontology languages can have various flavors and tastes. Some of them come from the domain of conceptual modelling (ER and UML – providing conceptual diagrams, Conceptual graphs – providing graphical interface for the first-order logic). Some have roots in formal logics: predicate logic, first-order logic (e.g., KIF, Knowledge Interchange Format), higher order logics (e.g., LBase), non-classical logics (e.g., F-Logic, Non-Mon, and modalities), description logics (e.g., OIL, DAML+OIL, OWL). Other languages are rule-based (such as RuleML, LP/Prolog), or are used to provide semantic annotations to the web resources and links between these resources (as SHOE, Simple HTML Ontology Extensions, RDF together

with RDFS). There exist also some projects and initiatives that influenced semantic knowledge modelling significantly (as, for example, Dublin Core and CYC).

OWL is an ontology language designed to facilitate ontology development and sharing via the Web, with the ultimate goal of making Web content more accessible to machines. It is the result of the Web Ontology Working Group (now closed) and descends from DAML+OIL, which is in turn an amalgamation of DAML and OIL.

9.2. OWL overview

OWL allows explicit, formal conceptualizations of knowledge models. It meets the following criteria set for ontology languages:

- the language should have a well-defined syntax, suitable for machine-processing, a formal semantics, and provide convenient way of writing expressions (XML syntax and the XML Schema datatypes are preferable);
- the language should derive from description logic theory and have sufficient expressive power, so reasoning systems would support it efficiently;
- the language should provide the notions of such abstracts as: concept (or class), role (or property), and individual;
- the language design should facilitate development of ontologies in a distributed fashion, with their versioning and reuse, and ontology, as well abstracts declared within its scope, should be identified with URI.

Thus, OWL provides constructs for properties, classes and relationships definitions. The OWL is partially mapped on the description logic, which is a subset of predicate logic, for which efficient reasoning support is possible. Ontology described in OWL can provide the answers to the questions: "What is x? Is this x?" Based on OWL ontology description one can run: consistency check ("Are there any contradictions in this model?"); classification (what are all the inferred types of this resource?"); satisfiability check ("Are there any classes in this ontology that cannot possibly have any members?").

OWL is built upon RDF and RDFS and has the same XML-based syntax. It replenishes RDF and RDF Schema missing features as: local scope of properties, disjointness of classes, Boolean combinations of classes, cardinality of restrictions, and special characteristics of properties. Since that, ontology in OWL is a collection of RDF triples with a specific OWL-defined meaning.

The original OWL specification became a World Wide Web Consortium (W3C) recommendation in February 2004, after almost three years of academic and industry development [27]. In October 2007, a new W3C working group started with the aim of extending OWL with several new features as proposed in the OWL 1.1 member submission (19 December 2006). W3C announced the new version, of the standard, OWL 2, on 27 October 2009 (see Table 9.5).

9.2.1. OWL vocabulary

OWL vocabulary consists of terms defined in owl namespace and uses some terms defined in `rdf`, `rdfs`, and `xsd` namespaces (Table 9.1). Three OWL sublanguages group and restrict OWL terms and thus have different expressiveness:

OWL Lite (Table 3.2) allows to build classification hierarchy and to declare simple constraints what is enough to create thesauri and simple ontologies. The constraints on range, existence and cardinality (0 or 1) together with opportunity to define equality and properties characteristic (inverse, transitive, and symmetric) reduce RDF expressiveness (OWL Lite

requires that URIrefs denoting classes, properties, and individuals to be mutually disjoint while the RDF data model imposes no limitations on URIrefs use).

OWL DL (Table 9.3) extends expressiveness of the language while retaining computational completeness and decidability (conclusions can be reached in finite time). The DL in the name of this sublanguage reflects its correspondence to description logics. The vocabulary of OWL DL includes all OWL language constructs but with some restrictions (negation, disjunction, cardinality, set operators, value restrictions, and enumerations). A class cannot also be an individual or property, a property cannot also be an individual or class.

OWL Full (Table 9.3) gives maximum expressiveness, however its use does not give computational guarantees (complete or efficient reasoning support cannot be provided). The vocabulary is the same as in DL case, but restrictions are released. Classes can be treated simultaneously as both collections and individuals, datatype properties can be marked as inverse functional. This sublanguage is fully compatible with RDF syntax and semantics. Every OWL (Full, DL, or Lite) document is an RDF/XML document; every RDF/XML document is an OWL Full document; not all RDF/XML documents are OWL DL (or OWL Lite) documents.

Table 9.1. Namespaces used in OWL.

prefix	namespace
`rdf`	`http://www.w3.org/1999/02/22-rdf-syntax-ns#`
`rdfs`	`http://www.w3.org/2000/01/rdf-schema#`
`xsd`	`http://www.w3.org/2001/XMLSchema#`
`owl`	`http://www.w3.org/2002/07/owl#`

Table 9.2. OWL Lite vocabulary.

RDF Schema Features:
`Class`
`(Thing, Nothing)`
`rdfs:subClassOf`
`rdf:Property`
`rdfs:subPropertyOf`
`rdfs:domain`
`rdfs:range`
`Individual`
	(In)Equality:
	`equivalentClass`
	`equivalentProperty`
	`sameAs`
	`differentFrom`
	`AllDifferent`
	`distinctMembers`
		Property Characteristics:
		`ObjectProperty`
		`DatatypeProperty`
		`inverseOf`
		`TransitiveProperty`
		`SymmetricProperty`
		`FunctionalProperty`
		`InverseFunctionalProperty`

Class Intersection:
`intersectionOf`
	Restricted Cardinality:
	`minCardinality (0 or 1)`
	`maxCardinality (0 or 1)`
	`cardinality (0 or 1)`
		Property Restrictions:
		`Restriction`
		`onProperty`
		`allValuesFrom`
		`someValuesFrom`

Header Information:
`Ontology`
`imports`
 	
Datatypes:
xsd datatypes
	Versioning:
	`versionInfo`
	`priorVersion`
	`backwardCompatibleWith`
	`incompatibleWith`
	`DeprecatedClass`
	`DeprecatedProperty`
		Annotation Properties:
		`rdfs:label`
		`rdfs:comment`
		`rdfs:seeAlso`
		`rdfs:isDefinedBy`
		`AnnotationProperty`
		`OntologyProperty`

Table 9.3. OWL DL and OWL Full vocabulary (addition or extension of OWL Lite).

Class Axioms:
`oneOf`
`DataRange`
`disjointWith`
`equivalentClass`
(applied to class expressions)
`rdfs:subClassOf`
(applied to class expressions)
	Boolean Combinations of Class Expressions:
	`unionOf`
	`complementOf`
	`intersectionOf`
	
	Filler Information:
	`hasValue`
		Arbitrary Cardinality:
		`minCardinality`
		`maxCardinality`
		`cardinality`

OWL 2 extends the original OWL vocabulary (Table 9.4), re-uses the same namespaces, and is backwards-compatible. The extensions were motivated by the feedback from the users of the original OWL version. The extensions obey: keys; property chains; richer datatypes and data ranges; qualified cardinality restrictions; asymmetric, reflexive, and disjoint properties; enhanced annotation capabilities. OWL 2 makes use of XML Schema facets while declaring restrictions.

Table 9.4. OWL 2 vocabulary (extension to the original OWL vocabulary).

Annotations	Properties
owl:annotatedProperty owl:annotatedSource owl:annotatedTarget owl:Annotation owl:AnnotationProperty owl:versionIRI owl:deprecated owl:priorVersion owl:versionInfo owl:backwardCompatibleWith owl:incompatibleWith	owl:assertionProperty owl:AsymmetricProperty owl:bottomDataProperty owl:bottomObjectProperty owl:onProperties owl:propertyChainAxiom owl:propertyDisjointWith owl:hasSelf owl:hasKey owl:topDataProperty owl:topObjectProperty owl:IrreflexiveProperty owl:ReflexiveProperty
Semantical extensions	**Extended datatype capabilities**
owl:Axiom owl:sourceIndividual owl:targetIndividual owl:targetValue owl:disjointUnionOf owl:AllDisjointProperties owl:AllDisjointClasses owl:NegativePropertyAssertion owl:NamedIndividual owl:hasValue owl:qualifiedCardinality owl:maxQualifiedCardinality owl:minQualifiedCardinality owl:members owl:onClass	owl:DataRange owl:datatypeComplementOf owl:onDataRange owl:onDatatype owl:withRestrictions **Facets** rdf:langRange xsd:length xsd:minLength xsd:maxLength xsd:minInclusive xsd:minExclusive xsd:maxInclusive xsd:maxExclusive xsd:totalDigits xsd:fractionDigits xsd:Pattern

OWL 2 releases some of the restrictions applicable to OWL DL and defines three new language profiles. Each profile is more restrictive than OWL DL, and consists of subset of structural elements defined in [25]:

OWL 2 EL – enables polynomial time algorithms for all the standard reasoning tasks,

OWL 2 QL – enables conjunctive queries to be answered in LogSpace (more precisely, AC0) using standard relational database technology,

OWL 2 RL – enables the implementation of polynomial time reasoning algorithms using rule-extended database technologies operating directly on RDF triples.

The OWL 2 specification consists of several normative and non-normative documents (see Table 9.5 for short description). The semantics of OWL 2 is defined based on RDF semantics [24] and logic style semantics (direct semantics) [16]. In addition to these, the specification provides the definition of a new, optional language syntax [17].

The OWL ontology can be viewed as a collection of RDF triples, but those triples that use the OWL vocabulary and have a specific OWL-defined meaning (see Figure 9.1). If a given RDF graph (or subgraph) instantiates the OWL specification, then OWL provides a semantic interpretation for the components of that graph or subgraph. Other portions of the RDF graph that do not follow the OWL specification have no OWL semantic interpretation – though, of course, they will have an RDF interpretation.

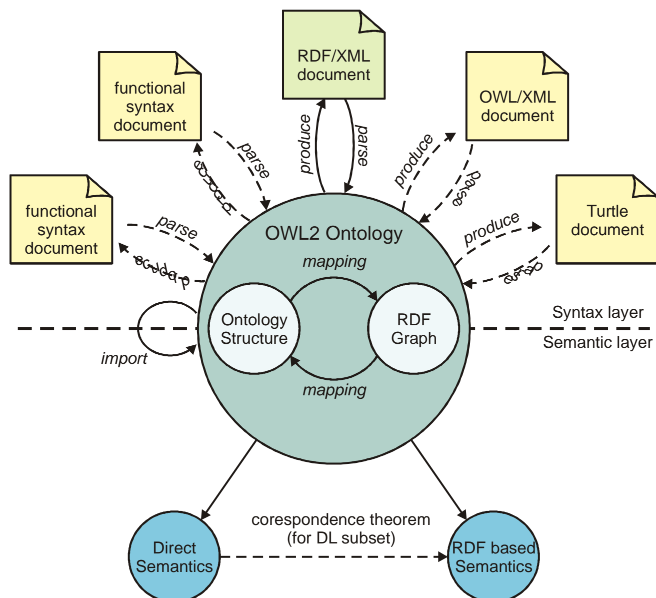


Figure 9.1. OWL2 ontology syntax, structure and semantics (based on http://www.w3.org/TR/2009/REC-owl2-overview-20091027/).

Table 9.5. OWL 2 specification.

	Type	Document
1	For Users	[19] gives quick overview of the OWL 2 specification.
2	Core Specification	[25] defines the constructs of OWL 2 ontologies in terms of both their structure and a functional-style syntax, and defines OWL 2 DL ontologies
3	Core Specification	[23] defines a mapping of the OWL 2 constructs into RDF graphs
4	Core Specification	[16] defines the meaning of OWL 2 ontologies in terms of a model-theoretic semantics.
5	Core Specification	[24] defines the meaning of OWL 2 ontologies via an extension of the RDF Semantics.
6	Core Specification	[14] provides requirements for OWL 2 tools and a set of test cases to help determine conformance.
7	Specification	[21] defines three sub-languages of OWL 2
8	For Users	[20] provides an approachable introduction to OWL 2
9	For Users	[18] provides an overview of the main new features of OWL 2 and motivates their inclusion in the language.
10	For Users	[22] provides a brief guide to the constructs of OWL 2 and changes from OWL 1.
11	Specification	[26] defines an XML syntax for exchanging OWL 2 ontologies
12	Specification	[17] defines an easy-to-read, but less formal, syntax for OWL 2
13	Specification	[15] specifies an optional extension to OWL 2

9.3. OWL details

There are few basic concepts used to describe ontologies: classes (sets of resources); relations between classes (hierarchy, disjointedness, etc.); individuals (member of at least one class); properties (used to describe resources). OWL specification provides notions for these concepts together with notions for cardinality, equality, property typing, characteristics of properties, and enumerated classes.

A normative serialization of OWL is RDF/XML. Thus, an OWL document is an XML documents with elements, tags, and namespaces. It starts with an ontology header and includes annotations, class and property definitions (axioms), descriptions of individuals, datatype definitions (that describe ranges of values). Object, datatype, and annotation properties in OWL must be disjoint. No URI can be typed as more than one kind of property. Classes and datatypes must be disjoint. No URI can be typed as both a class and a datatype. One URI can represent both a class and an individual.

9.3.1. OWL header

A header defines and describes the resource representing the ontology itself. In RDF/XML serialization the root element is identified as an `rdf:RDF` element with namespaces declarations. This element includes `owl:Ontology` element, containing typically: comments, labels, versioning information, and ontology import statements. The definitions of classes, object and datatype properties come next, see the example below.

RDF/XML serialization

```
<?xml version=''1.0'' encoding=''UTF-8''?>
xmlns:owl = "http://www.w3.org/2002/07/owl#"
xmlns:rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:rdfs = "http://www.w3.org/1999/02/22-rdf-schema#"
xmlns:xsd = "http://www.w3.org/2000/1/XMLSchema#">

<owl:Ontology rdf:about = "http://example.org/MyOntology.owl">
<owl:versionInfo>1.0</owl:versionInfo>
<owl:imports rdf:resource="http://example.org/AnotherOntology.owl"/>
</owl:Ontology>
</rdf:RDF>
```

Turtle serialization

```
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<http://example.org/MyOntology.owl> rdf:type owl:Ontology;
owl:imports "http://example.org/AnotherOntology.owl";
owl:versionInfo "1.0" .
```

The list of OWL vocabulary terms used in a header is following:

`owl:Ontology` – defines ontology

`owl:imports` – is used to specify those external ontologies that should be imported into this ontology

`owl:priorVersion` – provides information about prior version of this ontology

`owl:backwardCompatibleWith`– provides information about previous versions of this ontology that it is backward compatible with

`owl:incompatibleWith` –provides information about version of the ontologies this ontology is not compatible with

9.3.2. Classes

A class is a collection of individuals that share some characteristics as declared in the class definition. The class definition can consists of optional annotations, followed by constructs restricting the membership of the class. There are various forms of possible class restrictions. They can be used to set class hierarchy (subclass relationships), enumerate class membership explicitly, combine different classes (set operations as union-of, intersection-of, complement-of), declare disjointeness of memberships of classes.

The base class of all user-defined classes is `owl:Thing`. Every individual is a member of `owl:Thing` class. OWL also defines the empty class, `owl:Nothing`. Every class in OWL must be a member of `owl:Class`, and every resource that has an `rdf:type` of `owl:Class` is a class. In the following example `Female` is a class which is subclass of `Person` and is disjoint with `Male`.

```
:Female rdf:type owl:Class ;
        rdfs:subClassOf :Person ;
        owl:disjointWith :Male .
```

Individuals may become members of classes in two ways: by asserting their membership explicitly through typing (just as in RDF, using `rdf:type` term), and by deriving their membership based on their properties. The set of individuals that are members of a class is called the extension of this class. One individual can belong to more then one class extension. Two classes can have the same extension (until they differentiate by asserting new members).

NOTE

Individuals in OWL are not the same as individuals in object-oriented programming. These two concepts differ slightly. In OWL, individuals are member of the same class because they share some characteristics. The structure of each individual does not necessarily depend on the class the individual is a member of. In other words, class membership is decided based on the characteristics of individuals. In object-oriented programming, the characteristics of the class dictate the structure of its members.

In OWL Lite, subject of `rdfs:subClassOf` or `owl:equivalentClass` must be a class name and the object must be either a class name or a property restriction. OWL Lite does not support `owl:disjointWith`. There is no such restriction in OWL DL.

Using `owl:sameAs` to equate two classes is not the same as equating them with `owl:equivalentClass`. In OWL Full `owl:sameAs` may be used to equate anything a class and an individual, a property and a class, etc., and causes both arguments to be interpreted as individuals.

In the following example three classes, `Car1`, `Car2`, and `Car3` are defined. All three classes are disjoint. There are also three instances `c1`, `c2`, `c3` of the class `Car3` defined. All three are different.

RDF/XML

```
<!--  Classes  -->
<owl:Class rdf:about="Car1"/>
<owl:Class rdf:about="Car2"/>
<owl:Class rdf:about="Car3"/>

<!--  Individuals -->
<owl:NamedIndividual rdf:about="c1">
    <rdf:type rdf:resource="Car1"/>
</owl:NamedIndividual>
<owl:NamedIndividual rdf:about="c2">
    <rdf:type rdf:resource="Car1"/>
</owl:NamedIndividual>
<owl:NamedIndividual rdf:about="c3">
    <rdf:type rdf:resource="Car1"/>
</owl:NamedIndividual>

<!--  General axioms  -->
<owl:AllDifferent>
<owl:distinctMembers
rdf:parseType="Collection">
    <rdf:Description
rdf:about="c1"/>
    <rdf:Description
rdf:about="c2"/>
    <rdf:Description
rdf:about="c3"/>
</owl:distinctMembers>
</owl:AllDifferent>

<owl:AllDisjointClasses>
    <owl:members
rdf:parseType="Collection">
        <rdf:Description
rdf:about="Car1"/>
        <rdf:Description
rdf:about="Car2"/>
        <rdf:Description
rdf:about="Car3"/>
    </owl:members>
</owl:AllDisjointClasses>
```

Turtle

```
#    Classes
:Car1 rdf:type owl:Class .
:Car2 rdf:type owl:Class .
:Car3 rdf:type owl:Class .

#    Individuals
:c1 rdf:type :Car3 ,
      owl:NamedIndividual .
:c2 rdf:type :Car3 ,
      owl:NamedIndividual .
:c3 rdf:type :Car3 ,
      owl:NamedIndividual .

#    General axioms
[ rdf:type owl:AllDifferent ;
 owl:distinctMembers ( :c1 :c2 :c3 )
] .
[ rdf:type owl:AllDisjointClasses ;
 owl:members ( :Car1 :Car2 :Car3)
] .
```

Final forms of OWL serialization can be different. In the example below the definitions of three disjoint classes and three different instances are provided (it is assumed, that `owl` entity was declared before these definitions, as well classes and instances) which is semantically the same as the definitions given in the example above.

```
<!--  General axioms  -->
<rdf:Description>
    <rdf:type rdf:resource="&owl;AllDifferent"/>
    <owl:distinctMembers rdf:parseType="Collection">
        <rdf:Description rdf:about="c1"/>
        <rdf:Description rdf:about="c2"/>
        <rdf:Description rdf:about="c3"/>
    </owl:distinctMembers>
</rdf:Description>
<rdf:Description>
    <rdf:type rdf:resource="&owl;AllDisjointClasses"/>
    <owl:members rdf:parseType="Collection">
        <rdf:Description rdf:about="Car1"/>
```

```
        <rdf:Description rdf:about="Car2"/>
        <rdf:Description rdf:about="Car3"/>
    </owl:members>
</rdf:Description>
```

Basic terms for defining classes and their hierarchies:

`owl:Class` – a term used to declare a class

`owl:Thing` – the base class of all classes, all individuals belongs to this class

`owl:Nothing` – the empty class, its extension contains no individuals

`owl:subClassOf` – a transitive property used to assert that derived class is more specific than the base class, and that members of derived class are also members of base class

`rdf:type` – a property that declares the class of which an individual is a member

`owl:equivelentClass` – a property used to define equivalence of classes. Classes associated by this property have equivalent extensions.

Terms used to describe the membership of a class in terms of the extensions of other classes:

`owl:intersectionOf` – a property used for specifying that members of the class are members of all of the specified classes

`owl:unionOf` – a property used to declare that the members of this class are members of at least one of the specified classes

`owl:complementOf` – a property used for specifying that members of the class are not members of the specified class

`owl:sameAs` – a property that specifies that two individuals are the same individual (two URIrefs denote the same individual). Because in OWL Full classes and properties may be treated as individuals, in this sublanguge `owl:sameAs` may be used to indicate that two classes (or two properties) are indeed the same.

`owl:oneOf` – a property used to limit membership of the class to the specified collection of individuals. Thus the class can be defined by enumerating its instances, that is by declaring its extent explicitly.

`owl:AllDisjointClasses` – a class used with `owl:members` to specify a set of pair-wise disjoint classes. It is a shortcut for defining that a set of classes is pair wise disjoint using `owl:disjointWith`.

`owl:disjointWith` – a property used to declare that the class is disjoint with a given set of individuals. The memberships of two classes connected by this relation will share no individuals, thus no instance of either class can be an instance of both classes.

`owl:disjointUnionOf` – a property used to specify that this class is the union of the set of specified classes and that those classes are pair-wise disjoint. It is easy to assign this property to the class, however, any future attempts to incorporate a new subclasses will require to redefine the disjoint union to include these new subclasses.

`owl:AllDifferent` – a class used with `owl:distinctMembers` to define a collection of mutually distinct individuals. It is a shorthand for specifying multiple `owl:differentFrom` properties.

`owl:differentFrom` – a property used to declare that two individuals are different

9.3.3. Properties

The concept of property in OWL is similar to the concept of property in RDF. It is used to establish a binary relation between two resources. OWL specification defines two main types of properties: object properties (relationships between two individuals) and data properties (relationships between individuals and literals or XML Schema datatypes). They let to assert general facts about the members of classes and facts about individuals.

The properties in OWL are declared in a similar way as properties in RDF, with possible addition of some constrains. The basic declarations of properties starts with `owl:ObjectProperty` (for object properties) or `owl:DataProperty` (for data properties) and include `rdf:ID`, `rdfs:domain` and `rdfs:range`. Domain and range relationships are globally asserted. They specify the class memberships of individuals and the datatypes of literals. One property can be declared with multiple domains (or ranges). This means that the domain (or range) of this property is the intersection of the identified classes. Properties can be organized into hierarchies (using `owl:subPropertyOf`), can have their own characteristics (symmetric, reflexive, transitive, and so on), can be restricted, and can be associated with a key.

Declarations of properties' hierarchies and characteristics have a global scope. The property restrictions describe properties within the context of a specific class. They define how a property is to be used when it is applied to an instance of a particular class. In other words, a restriction describes the class of individuals that meet the specified property-based conditions.

Restrictions are declared using the construct `owl:Restriction`, and `owl:onProperty`, which identifies the property to which particular restriction applies.

Restrictions become related to the classes through the use of either `rdfs:subClassOf` or `owl:equivalentClass`. Restrictions have no names and must be defined using anonymous resources – they are relevant only in the context of the class in which they are defined and never need to be referred to.

Restriction defined using `rdf:subclassOf` specifies conditions necessary for the membership – all members of the class must meet the conditions specified by this restriction. Restriction defined using `owl:equivalentClass` specifies necessary and sufficient conditions to assert that an individual is a member of the class. Thus, with `owl:equivalentClass` definition, class members must meet the conditions of the restriction and any individual who meets the conditions of the restriction is implicitly a member of the class. Restrictions can also appear also as the objects of statements that combine classes using set operators.

A single class can contain many restrictions. When this is the case, each restriction is applied independently of the others to create a set of conditions that are either necessary or sufficient for membership in the class.

There are two kinds of property restrictions: value and cardinality. The value restrictions specify the range of a property when it is used with an instance of a particular class (owl:`all ValuesFrom`, `owl:someValuesFrom`, and `owl:hasValue`). There is one value restriction that is different from the others – `owl:selfRestriction`. This restriction, when applied to a property, refers to the class of all individuals that are related to themselves using this property.

Cardinality restrictions specify under very precise terms how many times a property can be used to describe an instance of a class (`owl:minCardinality`,

owl:maxCardinality etc.). Cardinality and value restrictions can be combined to create more complex conditions for class membership.

In addition, OWL 2 introduced the concept of qualified cardinality restrictions, which combine cardinality and value restrictions and allow to specify expected number of properties and their range. The restrictions of this kind can be constructed for datatype properties (using owl:onDataRange to identify a range of data values for the restriction) and object properties (using owl:onClass to identify the class to which the restriction refers.

A key describes a set of such properties, whose values uniquely identify the individuals. Thus keys are similar to inverse functional properties. The difference is that keys group one or more property expressions to uniquely identify the subject they describe. Keys are class-specific. A class becomes associated with a set of properties that are the keys for instances of that class after declaration of owl:hasKey property for it.

The following code shows how to declare classes using properties, how to apply restrictions and how to characterize properties.

```
:Person rdf:type owl:Class ;
    owl:equivalentClass [ rdf:type owl:Class ;
                            owl:unionOf ( :Female
                                          :Male )
                          ] ;
    rdfs:subClassOf [ rdf:type owl:Restriction ;
                          owl:onProperty :hasParent ;
                          owl:cardinality "2"^^xsd:nonNegativeInteger
                        ] .

:hasSibling rdf:type owl:ObjectProperty ,
                     owl:SymmetricProperty ;
            rdfs:range :Person .
```

Basic terms for defining properties and their hierarchies:

owl:ObjectProperty – the class of all properties that link two individuals.

owl:DatatypeProperty – the class of all properties that link an individual with a literal value.

owl:topObjectProperty – a property that connects all possible pairs of individuals.

owl:bottomObjectProperty – a property that connects no pairs of individuals.

owl:topDataProperty – a property that connects all possible individuals with all possible literals.

owl:bottomDataProperty – a property that does not connect any individual with a literal

rdfs:subPropertyOf – a property that specifies that one property is more specific than the other. If a property p1 is a subproperty of a property p2, the existence of a statement (A p1 B) implies the existence of the statement (A p2 B).

owl:propertyChain – a property used to build a chain of properties that represent the super property in a subproperty-of relationship. A property chain may consist of several, chained relationship, connecting two resources. Asserting that this chain is a subproperty of a property p1 has the consequence, that the existence of the property chain between two resources implies the existence of p1. Property chains can be used only as a part of a subproperty relationship and they can appear only in the subproperty position of such a relationship.

`rdfs:domain` – it is used to associate a property with a class (individuals with this property are members of the declared class)

`rdfs:range` – it is used to indicate the possible values for a property

Terms used to describe properties characteristics:

`owl:SymmetricProperty` – the class of all properties that are symmetric (such as "is sibling of", "equals", "adjacent to"). For any symmetric property p, the statement (A p B) implies the existence of the statement (B p A).

`owl:AsymmetricProperty` – the class of all properties that are explicitly not symmetric. For any asymmetric property p, the statement (A p B) implies the nonexistence of the statement (B p A).

`owl:ReflexiveProperty` – the class of all properties that are reflexive. For any reflexive property p and individual A, A is related to itself by p, (A p A).

`owl:IrreflexiveProperty` – the class of all properties that are not reflexive. For all irreflexive properties p and individuals A, there is no statement (A p A).

`owl:TransitiveProperty` – the class of all properties that are transitive (such as "has better grade than", "is taller than", "is ancestor of", etc.). For any transitive property p, (A p B) and (B p C) implies (A p C).

`owl:FunctionalProperty` – the class of all properties for which a given domain value has only a single range value (such as "age", "height", "directSupervisor", etc.). For any functional property p and individual A, (A p x) and (A p y) implies that x = y. `owl:FunctionalProperty` is a special subclass of the RDF class `rdf:Property`. Other property classes are subclasses of `owl:ObjectProperty`.

`owl:InverseFunctionalProperty` – the class of all properties for which a given object of a statement has only a single subject value, (A p B) and (C p B) implies that A = C (a property for which two different objects cannot have the same value).

`owl:inverseOf` – a property used to specify that two properties are inverse of each other. If a property p1 is inverse of a property p2, the existence of a statement (A p1 B) implies the existence of the statement (B p2 A). Both the domain and range of this relationship must be object properties. Datatype properties cannot have inverses because literal values cannot be the subjects of statements.

`owl:equivalentProperty` – a property used to assert that two properties are equivalent.

`owl:propertyDisjointWith` – Relationship that establishes that two properties (object properties or data properties) are disjoint. If two properties p1 and p2 are disjoint, it implies that no two statements with the same subject and object can have the predicates p1 and p2, so (A p1 B) and (A p2 B) both cannot holds.

`owl:AllDisjointProperties` – A class that is used with owl:members to describe collections of properties that are pair-wise disjoint. In addition to conventional, positive property assertions, OWL provides a notion of negative property assertions. Negative property assertions specify that a particular relationship does not exist between two individuals or between an individual and a literal value. The following vocabulary terms are used to specify negative property assertions:

`owl:NegativePropertyAssertion` – The class of all negative property assertions. In the most cases properties are used to specify relationships that exists. Negative properties assertions are used to express the fact that no relationship exists. Such assertions can be made

for both object and datatype properties. The specification requires to declare source individual, property, and target individual (when applied to object property) or target value (when applied to data property). Negative property assertions are instances of the class `owl:NegativePropertyAssertion` and cannot be named.

`owl:sourceIndividual` – the individual that is the subject of the negative property assertion.

`owl:assertionProperty` – a property that is the predicate of the negative property assertion.

`owl:targetIndividual` – the individual that is the object of the negative property assertion.

`owl:targetValue` – the literal value that is the object of the negative property assertion.

Terms used to build restrictions and keys:

`owl:Restriction` – the class of all restrictions. In general, in an OWL document `owl:Restriction` element contains `owl:onProperty` element and one or more restriction declarations. One restriction can define a range of values the property can take by using `owl:allValuesFrom`.

`owl:SelfRestriction` – the class of all self-restrictions. Self-restrictions identify classes of individuals related to themselves by a property.

`owl:onProperty` – a property that identifies the property to which a restriction applies.

`owl:allValuesFrom` – a property used to declare that all instances of the class must have values for this property only from the specified range.

`owl:someValuesFrom` – a property used to specify that all instances of this class must have at least one property with a value from the specified range.

`owl:hasValue` – a property used to specify that all instances of this class must have have an occurrence of this property with the specified value (to be a member of the class an instance must have at least one of its property values equal to the resource specified in `owl:hasValue`)

`owl:minCardinality` – a property used to specify that there must be at least N of the specified properties on each instance of this class. Cardinality expressions with values limited to 0 or 1 are part of OWL Lite. Positive integer values other than 0 and 1 are permitted in OWL-DL. In combination with `owl:maxCardinality` it limits the property's cardinality to a numeric interval.

`owl:maxCardinality` – a property used to specify that there must be at most N of the specified properties on each instance of this class.

`owl:cardinality` – a property used to specify that there must be exactly N of the specified properties on each instance of this class.

`owl:onClass` – a property used to identify the class of which the subject of a qualified cardinality restriction is a member.

`owl:minQualifiedCardinality` – a property used to specify that there must be at least N properties that have an individual of a particular class or value of a particular data range as the object of the statement.

`owl:maxQualifiedCardinality` – a property used to specify that there must be at most N properties that have an individual of a particular class or value of a particular data range as the object of the statement.

`owl:qualifiedCardinality` – a property used to specify that there are exactly N properties that having an individual of a particular class or value of a particular data range as the object of the statement.

`owl:hasKey` – a property used to identify a collection of properties that constitute a key for a given class.

9.3.4. Annotations

Annotations are primarily used in user interfaces. They describe classes, properties, individuals, datatypes and axioms in an ontology, including the ontology itself. Annotations are made by creating a statement that uses the annotation property as the predicate of this statement. The two most common annotation properties are `rdfs:label` and `rdfs:comment`.

Annotation properties have no semantic meaning, therefore they can not be specialized (using `owl:subPropertyOf`) nor characterized (using `owl:inversePropertyOf`), nor restricted by domain and range declarations. Introducing new annotation properties relies on declaring new instances of the class `owl:AnnotationProperty`.

Axiom annotations are slightly more complicated. Their exact structure depends on the type of axiom that is being described.

Terms used in annotation declarations:

`owl:AnnotationProperty` – the class of all annotation properties

`rdfs:label` – a property that provides a label representing a resource. It is often used by software tools as a substitution of resource's URI presented on user interface

`rdfs:comments` – a property that provides a textual description of the resource

`rdfs:seeAlso` – a property that specifies a resource that provides additional information

`rdfs:isDefinedBy` – a property that specifies a resource that defines the subject resource

`owl:deprecated` – a property that specifies whether or not the subject URI is deprecated

`owl:DeprecatedClass` – the class of all deprecated classes

`owl:DeprecatedProperty` – the class of all deprecated properties

`owl:versionInfo` – a property that provides information about the subject ontology or resource version

`owl:priorVersion` – a property that specifies a prior version of the ontology that is the subject of the statement

`owl:backwardCompatibleWith` – a property that specifies the URI of an ontology that is compatible with the ontology that is the subject of the statement

`owl:incompatibleWith` – a property that specifies the URI of an ontology that is not compatible with the ontology that is the subject of the statement

9.3.5. Datatypes and facets

Datatypes represent ranges of data values and are identified using URIs. OWL uses most of the built-in XML Schema datatypes (see Table 9.6) plus `rdfs:Literal` defined in RDF. All OWL reasoners are required to support the `xsd:integer` and `xsd:string` datatypes.

In addition to the predefined datatypes, OWL 2 introduced the ability to define custom datatypes by using facet restrictions or by defining them by means of other datatypes.

Table 9.6. Build-in XML Schema datatypes recommended for use with OWL (the other built-in XML Schema datatypes are problematic for OWL).

xsd:string xsd:normalizedString xsd:boolean
xsd:decimal xsd:float xsd:double
xsd:integer xsd:positiveInteger xsd:nonPositiveInteger xsd:negativeInteger xsd:nonNegativeInteger xsd:long xsd:unsignedLong xsd:int xsd:unsignedInt xsd:short xsd:unsignedShort xsd:byte xsd:unsignedByte xsd:hexBinary xsd:base64Binary
xsd:dateTime xsd:time xsd:date xsd:gYearMonth xsd:gYear xsd:gMonthDay xsd:gDay xsd:gMonth
xsd:anyURI xsd:token xsd:language xsd:NMTOKEN xsd:Name xsd:NCName

Facet restrictions restrict a set of valid values using a term defined in XML Schema namespace. To create a custom datatype with facet restriction one should create an instance of the class `rdf:Datatype` and associate one or more facet restrictions with it. In the example below an unnamed datatype was defined with a set of values allowed ranging from 0 (exclusive) up to 10 (inclusive).

```
[] rdf:type rdfs:Datatype;
owl:onDatatype xsd:integer;
owl:withRestrictions ([ xsd:minExclusive 0; ][ xsd:maxInclusive 3; ] ).
```

Custom datatypes in terms of other datatypes can be defined with the use of one of OWL set operator constructs: `owl:intersectionOf`, `owl:unionOf`, or `owl:datatypeComplementOf`. It is also possible to enumerate allowable values for the custom datatype with the use of `owl:oneOf` (see following two examples).

```
[] rdf:type rdfs:Datatype;
owl:unionOf (
[ rdf:type rdfs:Datatype; owl:onDatatype xsd:integer; owl:withRestrictions
( [ xsd:minExclusive 0; ] ) ]
[ rdf:type rdfs:Datatype; owl:onDatatype xsd:integer; owl:withRestrictions
( [ xsd:maxExclusive 3; ] ) ]
).

[] rdf:type rdfs:Datatype;
owl:oneOf ("1"^^xsd:int "2"^^xsd:int "3"^^xsd:int).
```

Note that the datatypes above are declared as b-nodes. The custom datatype definitions are usually provided within a definition of properties as in the following example (in Turtle and RDF/XML syntax):

```
:hasScore rdf:type owl:DatatypeProperty ;
   rdfs:range [ rdf:type rdfs:Datatype ;
                owl:oneOf ("1"^^xsd:int "2"^^xsd:int "3"^^xsd:int).
              ].

<owl:DatatypeProperty rdf:ID="hasScore">
 <rdfs:range>
 <owl:DataRange>
 <owl:oneOf>
 <rdf:List> <rdf:first rdf:datatype="&xsd;int">1</rdf:first>
  <rdf:rest>
   <rdf:List> <rdf:first rdf:datatype="&xsd;int">2</rdf:first>
    <rdf:rest>
     <rdf:List> <rdf:first rdf:datatype="&xsd;int">3</rdf:first>
      <rdf:rest rdf:resource="&rdf;nil" />
     </rdf:List>
    </rdf:rest>
   </rdf:List>
  </rdf:rest>
 </rdf:List>
 </owl:oneOf>
 </owl:DataRange>
 </rdfs:range>
</owl:DatatypeProperty>
```

Terms used to define custom datatypes:

`rdfs:Datatype` – the class of all datatypes.

`owl:onDatatype` – a property identifying the datatype to which the facet restrictions apply.

`owl:withRestrictions` – a property identifying a collection of facet restrictions that describe the datatype.

`owl:intersectionOf` – a property identifying a set of datatypes such that the datatype being described contains the values that are contained in all datatypes in the set.

`owl:unionOf` – a property identifying a set of datatypes such that the datatype being described contains any value that is contained in at least one of the datatypes in the set.

`owl:datatypeComplementOf` – a property specifying that the datatype being described contains all values that are not in the datatype that the property identifies.

`owl:oneOf` – a property identifying a set of values (values' enumeration) that make up the datatype.

The list of XML Schema facets supported in OWL 2:

`xsd:length` – declares the exact number of items (or characters) allowed.

`xsd:minLength` – declares the minimum number of items (or characters) allowed.

`xsd:maxLength` – declares the maximum number items (or characters) allowed.

`xsd:minInclusive` – declares the lowest value for the range of allowable values for the dataType being defined, including this value.

`xsd:minExclusive` – declares the lowest value for the range of allowable values for the dataType being defined, excluding this value.

`xsd:maxInclusive` – declares the highest value for the range of allowable values for the dataType being defined, including this value.

`xsd:maxExclusive` – declares the highest value for the range of allowable values for the dataType being defined, including this value.

`xsd:totalDigits` – declares the number of digits for the value.

`xsd:fractionDigits` – declares the maximum number of decimal places allowed.

`xsd:Pattern` – declares a regular expression that defines allowed character strings.

10. WSDL (Web Services Description Language)

A network service is a software component placed somewhere on the Internet, accessible via standard network protocols such as, but not limited to, SOAP over HTTP, providing its functionality through a standardized set of interfaces. A web service is based on XML technology, and is designed to support interoperable machine-to-machine interaction over a network. According to the SOA paradigm, the service provider should have an opportunity to register offered service in the network in order to provide possibility of service discovery and use to service clients. The service registration should record service description, that is, all information needed to start interaction with a service (including details about offered operations, used types, transport protocols, service location etc.).

A WSDL (Web Services Description Language) is an XML application, developed by W3C, used to write a formal, technical description of web service interfaces. A WSDL specification provides distinct definitions and terminologies used in web service description. It includes a schema for services declarations as collections of network endpoints, or ports, offering operations, and exchanging data through messages. The WSDL separates abstract definitions from their concrete uses or instances. This allows reusing the same parts of definitions in several services descriptions.

The WSDL description is enough to generate code partially, for a client or a server side of the service, in an automatic manner. It is also possible to generate WSDL description on a basis of existing code of web service implementation. There is no specific rule, what should be done first: service description or service implementation. All depends on a particular use case. The basic scenario of a WSDL service description use is following:

- a client discovers a web service in the network and analyses its description provided in the corresponding WSDL document,
- a client learns from the description about operations provided and data types used by the service,
- a client decides, which operations he or she is interested in, and builds the application code for it,
- if a client wants and can, it invokes chosen service's operations.

The code generation based on WSDL description can be done in a static way (when designing application from the beginning), or in a dynamic way (when creating a web service client or server instance during application execution). Although WSDL allows describing many types of network services, it is often used to describe SOAP-based services. In such case, a client program reads web service description and determines operations that are available from a corresponding WSDL document (usually generated automatically and provided on the same server that offers web service). Any specific datatypes used in the communication are declared in the WSDL document using XML Schema. Knowing all details, the client can use SOAP to actually call one of the discovered operations.

A WSDL is somewhat similar to an IDL (Interface Definition Language) used in CORBA. WSDL is a platform independent language, allowing service provider to describe all service's operations and principals of their invocation, and additionally, binding to the protocols, and the service's location.

There are two major versions of WSDL specification available: WSDL 1.1 (old W3C proposition, but still used), and WSLD 2.0 (current W3C recommendation). The WSDL 2.0 got its name after renaming WSDL 1.2 because of some substantial differences between 1.1 and 1.2 versions (the WSDL 1.2 was renamed to WSDL 2.0).

WSDL 2.0 accepts binding to all the HTTP request methods (not only GET and POST as in WSDL 1.1) so it better suits RESTful web services implementation. There is also SOAP 1.1 binding recommendation available. However, WSDL 1.1 has better support from software development tools. The major versions of WSDL specification, with their publication dates and URL addresses, are collected in the Table 10.1.

Table 10.1: Publication dates and URL addresses of major versions of WSDL specification (based on http://www.w3schools.com/w3c/w3c_wsdl.asp).

Specification	**Publication date**	**Status**	**Available at**
WSDL 1.1 Note	15 Mar 2001	Draft /Proposal	`http://www.w3.org/TR/wsdl`
WSDL Usage Scenarios	04 Jun 2002	Draft /Proposal	`http://www.w3.org/TR/ws-desc-usecases/`
WSDL Requirements	28 Oct 2002	Draft /Proposal	`http://www.w3.org/TR/ws-desc-reqs/`
WSDL Architecture	11 Feb 2004	Draft /Proposal	`http://www.w3.org/TR/ws-arch/`
WSDL Glossary	11 Feb 2004	Draft /Proposal	`http://www.w3.org/TR/ws-gloss/`
WSDL Usage Scenarios	11 Feb 2004	Draft /Proposal	`http://www.w3.org/TR/ws-arch-scenarios/`
WSDL 1.2 Core Language	11 Jun 2003	Draft /Proposal	`http://www.w3.org/TR/2003/WD-wsdl12-20030611`
WSDL 1.2 Message Patterns	11 Jun 2003	Draft /Proposal	`http://www.w3.org/TR/2003/WD-wsdl12-patterns-20030611/`
WSDL 1.2 Bindings	11 Jun 2003	Draft /Proposal	`http://www.w3.org/TR/2003/WD-wsdl12-bindings-20030611/`
WSDL 2.0 Primer	26 Jun 2007	Recommendation	`http://www.w3.org/TR/wsdl20-primer/`
WSDL 2.0 Core Language	26 Jun 2007	Recommendation	`http://www.w3.org/TR/wsdl20/`
WSDL 2.0 Adjuncts	26 Jun 2007	Recommendation	`http://www.w3.org/TR/wsdl20-adjuncts/`
WSDL 2.0 SOAP 1.1 Binding	26 Jun 2007	Recommendation	`http://www.w3.org/TR/wsdl20-soap11-binding/`
WSDL 2.0 RDF Mapping	26 Jun 2007	Recommendation	`http://www.w3.org/TR/wsdl20-rdf/`
Web Services Addressing Core	09 May 2006	Recommendation	`http://www.w3.org/TR/ws-addr-core/`
Web Services Addressing SOAP Binding	09 May 2006	Recommendation	`http://www.w3.org/TR/ws-addr-soap/`
Web Architecture	15 Dec 2004	Draft /Proposal	`http://www.w3.org/TR/webarch/`

NOTE:

The WSDL specification does not include any explicit rules or mechanisms for including semantic information in the description of web services. Therefore, two services can have similar descriptions (with the same syntax), while doing completely different things. To resolve this ambiguity “Semantic Annotations for WSDL and XML Schema” (SAWSDL) specification was developed by SAWSDL Working Group, as a part of W3C Web Services Activity. SAWSDL defines mechanisms, which relies on adding semantic annotations to WSDL components (see Chapter 11 for more details).

10.1. Structure of a WSDL document

A WSDL document is simply a set of definitions that describes a Web service in terms of messages it sends and receives. In general, the structures of WSDL 1.1 document and WSDL 2.0 document are similar. There is an abstract section and a concrete section, each built from some constructs (see Figure 10.1). Such design promotes reusability of description and separation of the independent design concerns.

In the WSDL 1.1 document:

– The abstract section includes: port types, messages and types constructs. Port types are abstract collections of supported operations. Operations refer to messages that are abstract descriptions of the data being exchanged. Operations and messages are bound to a concrete network protocol and message format.
– The concrete section includes: binding that defines the concrete protocol and data format specifications for a particular port type. Port definition associates a network address with a reusable binding. A collection of ports defines a service.

In the WSDL 2.0 document:

– The abstract section includes: interfaces and types constructs. An interface collects together supported operations. An operation associates a message exchange pattern with one or more messages. Messages are described using a type system (usually XML Schema) for defining bodies of inputs, outputs and faults.
– The concrete section includes: binding that specifies transport and wire format details for one or more interfaces. An endpoint associates a network address with a binding. A group of endpoints that implement a common interface defines a service.

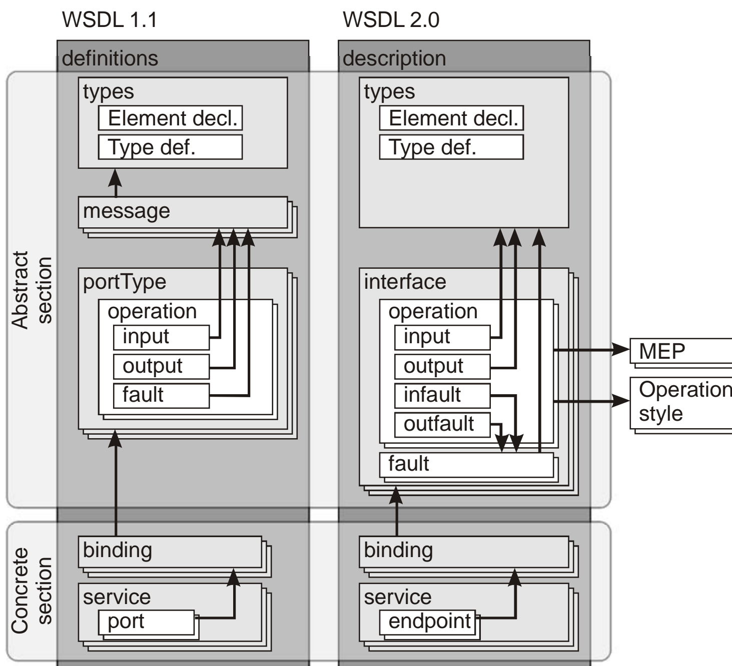


Figure 10.1. Comparison and analogies between WSDL 1.1 and 2.0 structures (based on http://en.wikipedia.org/wiki/File:WSDL_11vs20.png).

A short description of WSDL 1.1/WSDL 2.0 constructs is as follows.

`<definition>/<description>`: is a root element in a WSDL document that includes all WSDL constructs. WSDL allows extension elements within it. This element has been renamed to `description` in WSDL 2.0.

`<service>/<service>`: collects functions that have been exposed to the web based protocols. In WSDL 2.0, a service element can only implement one interface via its interface attribute. One service element can have more then one endpoint. If it happens, all endpoints employ different bindings and addresses for the same, shared interface (they provide the same behaviour via different binding configurations).

`<port>/<endpoint>`: defines the address or connection point to a web service, typically as a simple URL string. Port has been renamed to `endpoint` in WSDL 2.0.

`<binding>/<binding>`: contains binding declarations necessary to access the service (concrete protocol and data format specifications for the operations and messages).

`<portType>/<interface>`: defines a web service, operations that can be performed, and the messages that are used to perform the operation. The `<portType>` element has been renamed to `<interface>` in WSDL 2.0. WSDL 2.0 not only changed the element name, but also expanded the interface construct with a set of new elements and attributes, including an extends attribute that allows to declare interface inheritance (an interface that extends other interface offers all inherited operations plus the operations it defines directly).

`<operation>/<operation>`: defines constructs that can be compared to the methods or functions definitions known from programming languages. But operations are something more. An operation describes interaction with a service – so it does not only list function calls (messages exchanged between the service and its user or users) but also defines their number and order. This message exchange pattern is defined in the value of `pattern` attribute of an operation. WSDL2.0 uses new terms to describe message exchange patterns. For example, the "Request-Response" pattern from WSDL 1.1 has been renamed to the "In-Out" in WSDL 2.0, and the "One-Way" has been renamed to "In".

`<message>/N.A.`: Each service offers at least one operation which involves some communication between a requestor and service. This communication relies on sending queries and responses between both parts involved in form of messages. The order of queries (representing a call to a service) and responses (representing a service responses to the calls) must conform to an operation specific message exchanged pattern. Each message can consist of one or more logical parts which are description of the logical content of a message. Each part is associated with a message-typing attribute. A part may represent a parameter in the message; the bindings define the actual meaning of the part. Each message must have a unique name among all messages defined in a name attribute. Each part must have a unique name among all the parts of the enclosing message defined in the part name attribute. Messages have been removed in WSDL 2.0. Instead of defining message elements the WSLD designer simply refers to XML Schema types that defines bodies of inputs, outputs and faults.

`<types>/<types>`: is a place for descriptions of data types used in WSDL document (inline or referenced XML Schema types).

A WSDL document designer should be aware of not defining too many operations and messages. The rule is to define what is really needed and nothing more then that. Therefore, a WSDL document should not contain any parts that a client of the service does not need to know nor use. A WSDL document is not program code, it should rather be treated as a set of

metadata related to the code. To ensure platform independency a type system should be XML Schema based whenever possible.

10.2. Constructs in WSDL 1.1

WSDL 1.1 service description is an XML document including a set of information inside `<definitions>` root element. This element is a container for such elements as:

`<documentation>`, which provides a human readable description

`<import>`, which allows to include external definitions

`<types>`, which is a container for the data type definitions used to describe the content of the messages being exchanged.

`<message>`, which represents an abstract definition of data being transmitted during service operation execution. A message consists of logical parts, each of which is associated with a definition within some type system.

`<portType>`, which is an abstract set of operations. Each operation refers to input and output messages and can be supported by one or more endpoints.

`<binding>`, which specifies concrete protocol and data format for operations and messages defined by for a particular `portType`.

`<service>`, which is used to aggregate a set of ports definitions which specify a single communication endpoint as a combination of a binding and a network address.

10.2.1. Element <documentation>

Element documentation is used to provide a human readable description. This element can assists (be nested in) any other element appearing in a WSDL document.

10.2.2. Element <definitions>

This is a root element of any WSDL document. It works as a container for several fragments (nested elements) that forms together a full service description. It provides namespace declarations valid for its content along with XML namespace prefixes (see Table 10.2 for basic namespaces used in WSDL 1.1 documents).

Table 10.2: Basic namespaces and their prefixes that are used in WSDL documents.

Prefix	Namespace URI	Description
`wsdl`	`http://schemas.xmlsoap.org/wsdl/`	Namespace of WSDL grammar.
`soap`	`http://schemas.xmlsoap.org/wsdl/soap/`	Namespace of WSDL extension binding for SOAP message.
`http`	`http://schemas.xmlsoap.org/wsdl/http/`	Namespace of WSDL extension binding HTTP protocol.
`mime`	`http://schemas.xmlsoap.org/wsdl/mime/`	Namespace of WSDL extension binding for MIME protocol.
`soapenc`	`http://schemas.xmlsoap.org/soap/encoding/`	Namespace of schema governing SOAP 1.1 encoding.
`soapenv`	`http://schemas.xmlsoap.org/soap/envelope/`	Namespace of schema governing SOAP 1.1 envelopes.
`xsi`	`http://www.w3.org/2000/10/XMLSchema-instance`	Namespace of schema governing XML Schema instances. An instance is an XML document that conforms to a given XML Schema (.xsd) file.
`xsd`	`http://www.w3.org/2000/10/XMLSchema`	Namespace of schema governing XML Schema (.xsd) files.
`tns`	(application or context-dependent)	Convention used to refer to the current WSDL document. The prefix is an acronym for “this namespace.” Assigning the targetNamespace value to this prefix is customary.

The namespace declarations are provided in the `<definitions>` attributes with a `targetNamespace` attribute as a namespace declaration for the current document. The informal grammar of `<definitions>` element is following [52]:

```
<definitions name="nmtoken"? targetNamespace="uri" xmlns=.... .....?>
    <import namespace="uri" location="uri"/>*
    <documentation .... /> ?
    <types> ?
        <documentation .... />?
        <xsd:schema .... />*
        <-- extensibility element --> *
    </types>
    <message name="nmtoken"> *
        <documentation .... />?
        <part name="nmtoken" element="qname"? type="qname"?/> *
    </message>
    <portType name="nmtoken">*
        <documentation .... />?
        <operation name="nmtoken">*
           <documentation .... /> ?
           <input name="nmtoken"? message="qname">?
               <documentation .... /> ?
           </input>
           <output name="nmtoken"? message="qname">?
               <documentation .... /> ?
           </output>
           <fault name="nmtoken" message="qname"> *
               <documentation .... /> ?
           </fault>
        </operation>
    </portType>
    <binding name="nmtoken" type="qname">*
        <documentation .... />?
        <-- extensibility element --> *
        <operation name="nmtoken">*
           <documentation .... /> ?
           <-- extensibility element --> *
           <input> ?
               <documentation .... /> ?
               <-- extensibility element -->
           </input>
           <output> ?
               <documentation .... /> ?
               <-- extensibility element --> *
           </output>
           <fault name="nmtoken"> *
               <documentation .... /> ?
               <-- extensibility element --> *
           </fault>
        </operation>
    </binding>
    <service name="nmtoken"> *
        <documentation .... />?
        <port name="nmtoken" binding="qname"> *
           <documentation .... /> ?
           <-- extensibility element -->
        </port>
        <-- extensibility element -->
    </service>
    <-- extensibility element --> *
</definitions>
```

where ? denotes zero or one occurrence, * means zero or more occurrences; + means one or more occurrences; the dots .... stands for omitted information, irrelevant to the context; <-- extensibility element --> is a part with elements from some "other" namespace (like ##other in XSD). The diagram in Figure 10.2 illustrates the contents of this element (as defined in the http://schemas.xmlsoap.org/wsdl/, with some elements collapsed and without elements' attributes).

Figure 10.2. Diagram showing the content of the <definitions> element (with some nested elements collapsed and without elements' attributes, as defined in http://schemas.xmlsoap.org/wsdl/).

10.2.3. Element <import>

This elements works like the #include pre-processor directive in the C programming language. It allows splitting the description into independent documents and merging them in one, main document as necessary. This improves the modularity and legibility definition of the service. The import has two attributes: namespace and location.

10.2.4. Element <types>

This element is a container for data types definitions used in <message> elements. It contains zero or more sub-elements <schema>, which must adhere to the rules for XML Schema documents. The declared types can be complexType or simpleType.

10.2.5. Element <message>

This element defines a format of messages exchanged between a client and a web service. It may represent a query, response or error signal. It refers to data types defined in `<types>`. The data contained in `<message>` are abstract. A message consists of one or more sub-elements `<part>`.

Each `<part>` identifies portion of data exchanged in a message, and used data types. Typically, when client invoke an operation, it sends a message with input data. It receives back a response with output data. The order of `<part>` elements reflects the order of parameters of the operation being invoked.

10.2.6. Element <portType>

This element specifies a set of operations supported by the service endpoint (it provides a unique identifier to a group of operations supported by a single endpoint). Each `<operation>` is defined individually.

`<operation>` is an abstract definition of an operation supported by a Web service. The use of this element is analogous to the method declaration in Java, but with a set of messages representing method invocation and method results. Thus, there can be several input, output and fault messages defined. Input, output and fault messages are defined using nested `<input>`, `<output>` and `<fault>` elements. These elements refer to `<message>` elements defined in the same WSDL document or imported from external documents.

The order of messages should follow so called Message Exchange Patterns (MEP). WSDL 1.1 supports four MEPs:

- One-way - a port receives a message (operation consists of a single `<input>` child element).
- Request-response - a requester sends a message and receives reply from a port (operation consists of `<input>`, `<output>`, and optional `<fault>` child element).
- Notification - a port sends a message (there is only one message sent in the operation: `<output>`).
- Solicit-response - the port sends a notification and receives a response (operation consists of an `<output>`, `<input>`, and optional `<fault>` child element)

Each operation element must have a unique name, assigned to its name attribute, for all operations in the `<portType>`. Similarly, `<input>` and `<output>` elements must have unique names in the scope of `<portType>` defined assigned to their name attributes.

10.2.7. Element <binding>

This elements is used to specify a concrete protocol binding and data encoding for a given `<portType>` (i.e. it provides binding to HTTP, SOAP MIME or, possibly, custom protocols). Since in the WSDL document `<operation>` elements are already defined, the element `<binding>` maps the abstract definitions of operations, their input and output messages, to the appropriate protocol used by a web service.

10.2.8. Element <service>

This element appears typically in the end of the WSDL document. It defines a concrete web service endpoint with URL to the service location (there is no other occurrence of such URL before service element). `<service>` element groups one or more `<port>` elements. A single `<port>` element represents an endpoint (access point) to a web service.

10.3. Sample of WSDL 1.1 document

The WSDL 1.1 sample document provided below was generated using Eclipse IDE for SOA Developers. It contains a description of `HelloService` offering one `Operation` with `OperationRequest` as input, and `OperationResponse` as output, and SOAP binding.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<wsdl:definitions        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://www.example.org/HelloService/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"            name="HelloService"
targetNamespace="http://www.example.org/HelloService/">
  <wsdl:types>
    <xsd:schema targetNamespace="http://www.example.org/HelloService/">
      <xsd:element name="RequestMsg">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="in" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="ResponseMsg">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="out" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:schema>
  </wsdl:types>
  <wsdl:message name="RequestMsg">
    <wsdl:part element="tns:RequestMsg" name="parameters"/>
  </wsdl:message>
  <wsdl:message name="ResponseMsg">
    <wsdl:part element="tns:ResponseMsg" name="parameters"/>
  </wsdl:message>
  <wsdl:portType name="HelloService">
    <wsdl:operation name="Operation">
      <wsdl:input message="tns:RequestMsg" name="Request"/>
      <wsdl:output message="tns:ResponseMsg" name="Response"/>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="HelloServiceSOAP" type="tns:HelloService">
    <soap:binding                                          style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="Operation">
      <soap:operation
soapAction="http://www.example.org/HelloService/Operation"/>
      <wsdl:input name="Request">
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output name="Response">
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="HelloService">
    <wsdl:port binding="tns:HelloServiceSOAP" name="HelloServiceSOAP">
      <soap:address location="http://www.example.org/"/>
```

```
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

10.4. Constructs in WSDL 2.0

WSDL 2.0 description is an XML document conforming to the WSDL 2.0 specification and XML Schema (not all restrictions described in the specification can be defined in XML Schema). A WSDL document describes a set of Web service by providing its interface and a set of endpoint definitions. An interface consists of a set of operations. An operation is a sequence of input and output messages. An endpoint is defined by its address and binding. An address tells where the service is provided and binding tells how messages should be formatted and transmitted. Accompanying specification defines extensions of SOAP 1.2, SOAP 1.1, HTTP GET, HTTP POST, and MIME bindings.

The elements of WSDL 2.0 documents are defined in the `http://www.w3.org/ns/wsdl` namespace, and their XML schema is available as `http://www.w3.org/2002/ws/desc/ns/wsdl20.xsd` file.

10.4.1. Element <description>

The `<description>` is a main element of a WSDL document. The content of this element should conform to the following pattern:

```
<description
      targetNamespace="xs:anyURI" >
  <documentation />*
  [ <import /> | <include /> ]*
  <types />?
  [ <interface /> | <binding /> | <service /> ]*
</description>
```

where: `?` denotes an optional element (zero or more occurrences), `*` means zero or more occurrences, `+` means one or more occurrences, the brackets `[`and `]` are used for grouping, and the `|` represents an alternative choice.

Element `<description>` is defined in the namespace `http://www.w3.org/ns/wsdl`, has a mandatory attribute `targetNamespace` of type `anyURI`, and can have zero or more attributes defining other namespaces. The diagram in Figure 10.3 illustrates contents of this element (as defined in the `wsdl20.xsd`, shown with elements collapsed and without elements' attributes).

10.4.2. Element <documentation>

A human readable description in the WSDL 2.0 documents is provided within optional `<documentation>` elements. These elements can appear in all elements included in a `<description>`. The `<documentation>` syntax is following:

```
<documentation>
  [extension elements]*
</documentation>
```

10.4.3. Elements <include> and <import>

Element `<include>` allows to include components defined in other WSDL documents in the current interface definition. This allows describing a network of services in a modular manner - fragments once prepared can be used many times in several descriptions of various network services.

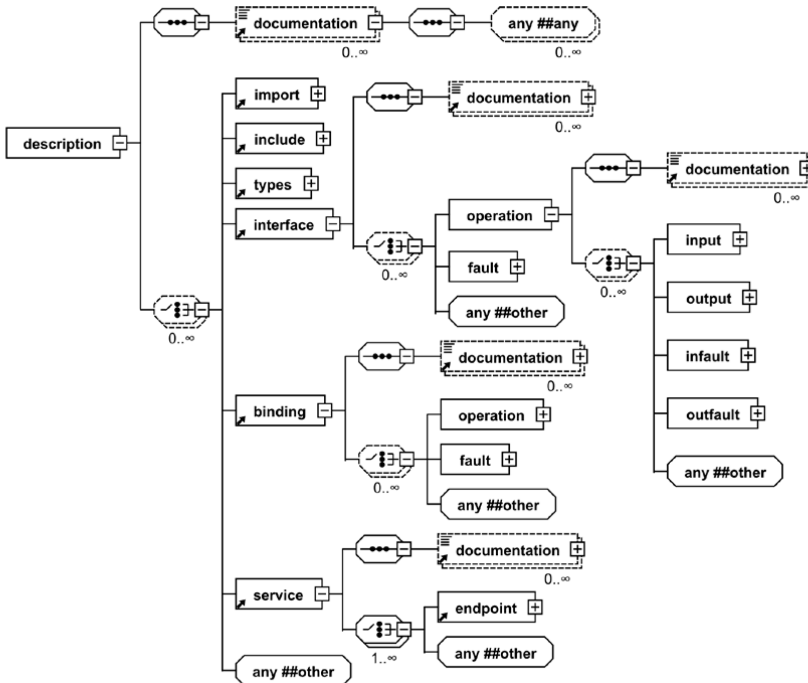


Figure 10.3. Diagram showing the content of the `<description>` element (with some nested elements collapsed and without elements' attributes, as defined in http://www.w3.org/2007/06/wsdl/wsdl20.xsd).

Element `<include>` has a mandatory location attribute which specifies a location of the external, included WSDL documents. The target namespace of attached WSDL descriptions must match the target namespace of the base document. Standard content of this element is as follows:

```
<description>
  <include
        location="xs:anyURI" >
    <documentation />*
  </include>
</description>
```

Element `<import>` has a similar meaning to `<include>`. The difference is that the imported WSDL document can have different target namespace than the base document. This element has a mandatory namespace attribute for an imported element and an optional location attribute. Standard content of this element is as follows:

```
<description>
  <import
        namespace="xs:anyURI" location="xs:anyURI"? >
    <documentation />*
  </import>
</description>
```

10.4.4. Element <types>

Element `<types>` serves as a place for definitions of data types used by exchanged messages. XML Schema is preferred as a typing language, although it is possible to use also the DTD or RELAX NG. The use of custom schemas relies on importing them (with the use of `<xs:import>`) or embedding them within `<types>` element of the WSDL document (using `<xs:schema>`). The elements from imported or included schemas can be referenced using their `QName`. WSDL allows using built-in types defined in XML Schema. Standard content of `<types>` is as follows:

```
<description>
  <types>
    <documentation />*
    [ <xs:import namespace="xs:anyURI" schemaLocation="xs:anyURI"? /> |
      <xs:schema targetNamespace="xs:anyURI"? /> |
      other extension elements ]*
  </types>
</description>
```

10.4.5. Element <interface>

Element `<interface>` contains `<operation>` elements, which group definitions of sequences of messages sent to and from the service. In other words: an operation is a sequence of input and output messages, and an interface is a set of operations. The operations can be defined in the interface directly or can be inherited from other interfaces that the current interface extends. Extending interface is optional. One interface can extend one or more other interfaces. To prevent loops in the definitions, the current interface must not appear in a set of interfaces it extends (either directly or indirectly).

The declaration of `<interface>` is equivalent to the definition in IDL or to the definition of an abstract class in object-oriented programming languages. `<interface>`, apart from `<operation>`, may contain `<fault>` elements with errors descriptions. The `<interface>` has mandatory attribute `name` (the value of which is a name of the interface), and two optional attributes: `extends` (which lists the interfaces that the interface extends) and `styleDefault` (which contains the default style used to create element declarations for all interface operations).

The `styleDefault` attribute is optional. It can be used to set a default value for the style attributes of all interface's operations. If it is missing, it simply means that no additional rules need to be followed. The value of `styleDefault` attribute can be overwriten by a style attribute of operation elements (individually for each operation as described below). The provided, predefined style values with short description are listed in Table 10.3. Details are provided in [50] specification.

Table 10.3: Predefined styles.

IRI	Description
`http://www.w3.org/ns/wsdl/style/rpc`	Requires that all operations within an interface must follow the rules for RPC-style messages
`http://www.w3.org/ns/wsdl/style/iri`	Places restrictions on message definitions so they may be serialized into something like HTTP URL-encoded
`http://www.w3.org/ns/wsdl/style/multipart`	In the HTTP binding, for XForms clients, a message must be defined following this style and serialized as "Multipart/form-data"

The content of <interface> should conform to the following pattern:

```
<description>
  <interface
        name="xs:NCName"
        extends="list of xs:QName"?
        styleDefault="list of xs:anyURI"? >
    <documentation />*
    [ <fault /> | <operation /> ]*
  </interface>
</description>
```

Element <fault>

```
<description>
  <interface>
    <fault
          name="xs:NCName"
          element="union of xs:QName, xs:token"? >
      <documentation />*
    </fault>
  </interface>
</description>
```

Element <fault> serves as an abstract description of a fault that MAY occur during invocation of an operation of an interface. The two attributes, name and element, are used, respectively, to declare a name of the fault and indicate contents of the fault message. Once declared, the fault element can be referred by name in multiple operations using ref attributes of their infault and outfault elements. The name attribute is required and must be unique within the parent interface element. The element attribute is optional and indicate a schema for the content or payload of the fault message (defined in the types section).

Element <operation>

Element <operation> describes interaction with a service by listing messages exchanged between the service and its users (in normal situation or caused by an error). Thus, the <operation> element may contain such elements as: <input>, <output>, <infault>, and <outfault>. The list of attributes of the <operation> consists of the following items: style, safe, pattern (all three are optional), and name (required). The content of <operation> should conform to the following pattern:

```
<description>
  <interface>
    <operation
          name="xs:NCName"
          pattern="xs:anyURI"?
          style="list of xs:anyURI"? >
      <documentation />*
      [ <input /> | <output /> | <infault /> | <outfault /> ]*
    </operation>
  </interface>
</description>
```

The value of style attribute overrides the value of styleDefault attribute of a parental <interface> element. Attribute style provides some additional information related to the operation, such as structural restriction on appearance of elements in a normal message or in an error message. For example, the value http://www.w3.org/ns/wsdl/style/iri indicates that a model of the message content must be defined as a complexType containing a sequence (these are types

defined in XML Schema). Using other structures (like choice) instead of sequence is prohibited. The complexType must have no attributes, and child elements of the sequence should inherit from simpleType, but cannot inherit or be of the type QName, NOTATION, hexBinary or base64Binary (the types defined in XML Schema). The number of elements in the sequence is not limited. The sequence may include local child elements, each of which may contain a nillable attribute. The localPart belonging to the QName of elements must be the same as the name used in the <interface> element.

The value of a pattern attribute represents a message exchange pattern, i.e. message placeholders, the order and number of messages involved in the interaction, the interaction's participants and their roles (source or sink of the messages). The WSDL specification neither define a machine understandable language for defining message exchange patterns, nor provide any specific patterns definitions. Some specific patterns are defined in [50] together with their IRIs that may be used as the value of the pattern attribute. The examples of patterns for the in-bound type of communication are following:

– In-Only – the pattern consists of only one message sent to the service by a node. Error message can not be generated.
– Robust In-Only – is a variation of In-Only. It contains, similarly, only one message sent to the service, but in this case an error messages can be generated;
– In-Out – this pattern consists of exactly two messages: a message received by a service from a node followed by a response message sent to a node by a service. The second message can be normal or an error message;
– In-Optional-Out - in this pattern one or two messages may appear: message received by a service from a node, followed by an optional response message sent to a node by a service. Each "in" message may trigger a response in the form of an error message.

The value of safe attribute, which can be "true" or "false", indicates whether an operation is considered to be "safe" or not. An operation is considered to be safe, if it does not permanently alter any part of the service environment or if it does not give the consumer any new obligations (as it is for "read only" operations). The default value is "false". Therefore, if the attribute is not present or if it is explicitly set to false, then the consumer can assume the operation is "not safe". The safe attribute is described in [50] specification.

The value of obligatory name attribute is used to declare an operation's name, which is used as its identifier.

Message containers are nested in the <operation> element. Their number and order depends on message exchange pattern used. There are <input> and <output> messages containers for normal messages, and <infault>, <outfault> containers for error messages. Although local names of these containers indicate whether a given message is incoming or outgoing, their accurate role must be defined in their messageLabel attribute (for example messageLabel="In"). The order of the input and output elements should be the same as the order defined in the message exchange pattern used.

Elements <input> and <output>

Elements <input> and <output> are message containers used in normal communication. Message exchange pattern includes rules on how to generate messages with information about the error. Elements <infault> and <outfault> are used as containers for messages with errors. Signaling an error might terminate message exchange pattern. The value of their mandatory messageLabel attribute indicates the role of messages they carry. This value must match a name defined in the message exchange pattern, and be consistent

with the direction stored in `token` (which may take one of the two values - `in` or `out`). A container can be marked as optional in a given message exchange pattern. A node that communicates with a service and a service both use these containers. The content of the message container should conform to the following pattern:

```
<description>
  <interface>
    <operation>
      <input
            messageLabel="xs:NCName"?
            element="union of xs:QName, xs:token"? >
        <documentation />*
      </input>
      <output
            messageLabel="xs:NCName"?
            element="union of xs:QName, xs:token"? >
        <documentation />*
      </output>
    </operation>
  </interface>
</description>
```

A required model of message content described by an `xs:token` can take one of the following values `#any`, `#none`, `#other`, or `#element`. The value `#any` indicates that the content of message is a single element. The value `#none` means no message content. The value `#other` indicates that the message content is described by some other properties of the extension indicating declarations in the system different from the XML type system extensions. The value `#element` indicates that the message consists of a single element described by the global element declaration referred in the declaration of that element. This property is used only when a message is described using a data model based on XML.

Declaration of an `element` is optional. This `element` represents a message content or a payload. If the content model has the value `#any` or `#none`, the element's declaration must remain empty.

10.4.6. Element <binding>

Element `<binding>` contains binding declarations necessary to access the service. These declarations describe a message format and transmission protocol for communication with an endpoint (including encoding of input and output parameters implemented in the service). The relevant declarations are described in the mandatory element `<operation>`. Once defined a `<binding>` can be used many times in the definitions of various interfaces. The service can support several bindings for a single interface, but each bound (endpoint) should be accessible under a unique address, identified by an URI. The syntax of the `<binding>` is as follows:

```
<description>
  <binding
        name="xs:NCName"
        interface="xs:QName"?
        type="xs:anyURI" >
    <documentation />*
    [ <fault /> | <operation /> ]*
  </binding>
</description>
```

10.4.7. Element <service>

Element `<service>` contains a set of `<endpoint>` elements, determining places where the service has been implemented. An `interface` attribute of the service contains the name of the interface, whose endpoints are just being described. The syntax of `<service>` declaration is as follows:

```
<description>
  <service
        name="xs:NCName"
        interface="xs:QName" >
    <documentation />*
    <endpoint />+
  </service>
</description>
```

Element <endpoint>

Element `<endpoint>` is used to define an endpoint. Two of its attributes - `name` (name of the endpoint) and `binding` (representing binding) are required. The third attribute, `address` (address of the service that implements the interface) is optional. The syntax of the endpoint declaration is as follows:

```
<description>
  <service>
    <endpoint
          name="xs:NCName"
          binding="xs:QName"
          address="xs:anyURI"? >
      <documentation />*
    </endpoint>+
  </service>
</description>
```

10.5. Sample of WSDL 2.0 document

The WSDL 2.0 file provided below includes a description of the InformerService. The service interface, `Informer`, has only one operation `GetOrderPrice`. The request should include the information on the product name and the number of ordered pieces. Then, in the response, the service will provide the total cost of the order.

```
<?xml version="1.0">
<description
  xmlns="http://www.w3.org/ns/wsdl"
  targetNamespace="http://example.com/wsdl20/informer"
  xmlns:tns="http://example.com/wsdl20/informer"
  xmlns:skl="http://example.com/schemas/informer"
  xmlns:wsoap="http://www.w3.org/ns/wsdl/soap" >

  <types>
    <schema targetNamespace="http://example.com/schemas/informer"
            xmlns="http://www.w3.org/2001/XMLSchema">
      <element name="OrderPriceRequest">
        <complexType>
          <all>
            <element name="productName" type="string"/>
            <element name="quantity" type="int"/>
          </all>
        </complexType>
      </element>
```

```
      <element name="OrderPriceResponse"
        <complexType>
          <all>
            <element name="price" type="float"/>
          </all>
        </complexType>
      </element>
    </schema>
  </types>

  <interface name="Informator" >
    <operation name="GetOrderPrice"
               pattern="http://www.w3.org/2004/03/wsdl/in-out">
      <input messageLable = "In"
             element="skl:OrderPriceRequest"/>
      <output messageLable = "Out"
             element="skl:OrderPriceResponse"/>
    </operation>
  </interface>

  <binding name="InformerSOAP" interface="tns:Informer"
   type="http://www.w3.org/ns/wsdl/soap"
   wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP/"
   wsoap:mepDefault="http://www.w3.org/2003/05/soap/mep/request-response">

     <operation ref="tns:GetOrderPrice">
         wsoap:action="http://example.com/GetOrderPrice"/>
     </operation>
  </binding>

  <service name="InformerService"  interface="tns:Informer">
     <documentation> Service test </documentation>
     <endpoint name="InformerPort"
               binding="tns:InformerSOAP"
               address ="http://example.com/informer"/>
     </endpoint>
  </service>
</description>
```

11. SAWSDL (Semantic Annotations for WSDL and XML Schema)

The SAWSDL is a W3C specification that describes a method for adding semantic annotations to various parts of a WSDL document. There are two basic types of annotations defined: *the model reference* and *the schema mapping*. The model reference associates WSDL or XML Schema component with a concept from a given ontology. Such concept can support a semantic based web service discovery. The schema mapping specifies the data mapping of XML Schema types to and from a semantic model. Thus, this is a description of transformation of data from one representation into another, which is especially applicable when trying to chain web services.

The SAWSDL annotation, in general, relies on using a set of extension attributes holding one or more URIs. Each URI typically refers to a concept in a semantic model that is external or internal to the WSDL document. This approach fits within the WSDL 2.0 [51], WSDL 1.1 [52] and XML Schema [61] extensibility frameworks.

The SAWSDL specification reached W3C Recommendation status on August 28, 2007. It derives from the earlier work of the semantic Web community OWL-S [28] and the other efforts METEOR-S [13], WSMO [55]. Many of SAWSDL ideas originate from the earlier specification WSDL-S [53].

The SAWSDL is not any new ontology definition language or mapping definition language, nor does depend on such languages. The possible choices of such languages are not restricted in any way.

The SAWSDL address the topic of “Semantic Web Services”. There exist some alternatives for describing Web Services semantically. It is SWSI (Semantic Web Services Initiative), incorporating the SWSL (Semantic Web Services Language) and the SWSO (Semantic Web Services Ontology). For more information on SWSI, please refer to http://www.ai.sri.com/daml/services/.

NOTE:

The **Semantic Web** is an extension of the current World Wide Web. It enables interpretation of published information not only web browsers users (humans), but also by machines that recognize and understand the semantics of web resources from the meta-information they are annotated with.

The meta-information gives the clue on what the annotated resources are about, how to interpret them, etc. The meta-information is well structured, and machine processable data, created using formal semantic models (ontologies). Two main W3C Standards commonly used for defining such models are RDF and OWL. Describing them briefly: RDF is used to represent information and to exchange knowledge, and OWL is used to define ontologies, supporting web searches and knowledge management (refer to Chapter 8 and Chapter 9 for details).

The **Semantic Web Services** are web services of the semantic web. These services make use of semantic annotations and markups to publish structured, machine processable data and to discover and interact with other services in a detailed and sophisticated way (including dynamic discovery, composition and invocation of services).

Machine processable semantic models and concepts are perfect for service description and discovery. Their applications run beyond simple keywords search limits, and assure receiving query matches that fits the desired functionality.

11.1. Annotation Mechanism

Extending web service description with semantic annotations in the SAWSDL relies on adding extension attributes to chosen WSDL or XML Schema components. The XML Schema of such extension attributes is following:

```
<xs:schema
targetNamespace="http://www.w3.org/ns/sawsdl"
xmlns="http://www.w3.org/ns/sawsdl"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:wsdl="http://www.w3.org/ns/wsdl">
<xs:simpleType name="listOfAnyURI">
<xs:list itemType="xs:anyURI"/>
</xs:simpleType>
<xs:attribute name="modelReference" type="listOfAnyURI" />
<xs:attribute name="liftingSchemaMapping" type="listOfAnyURI" />
<xs:attribute name="loweringSchemaMapping" type="listOfAnyURI" />
<xs:element name="attrExtensions">
<xs:complexType>
<xs:annotation>
<xs:documentation>This element is for use in WSDL 1.1 only. It does not apply to WSDL 2.0 documents. Use in WSDL 2.0 documents is invalid.</xs:documentation>
</xs:annotation>
<xs:anyAttribute namespace="##any" processContents="lax" />
</xs:complexType>
</xs:element>
</xs:schema>
```

All three extension attributes (`modelReference`, `liftingSchemaMapping`, `loweringSchemaMapping`) are of the `listOfAnyURI` type. It means that each extension attribute in a WSDL or XMLSchema can have a list of URIs assigned as its value. The form of the list is the same in all three cases, but its interpretation is different. In case of the first extension attribute, named `modelReference`, each URI on the list should identify (be a direct reference of) an ontological concept with which an annotated element should be associated. In case of two other extension attributes, named `liftingSchemaMapping` and `loweringSchemaMapping`, each URI on the list should identify a mapping between semantic data and XML. The lifting schema transforms from XML to semantic data, the lowering schema transforms from semantic data to XML. Multiple schema mappings should be treated as alternatives whereas multiple model references should all apply. SAWSDL does not specify any other relationship.

11.1.1. Model Reference

The `modelReference` attribute can be used with every element within WSDL and XML schema. However, the SAWSDL specification defines its use only to annotate:

XML Schema complex type definitions, simple type definitions, element declarations, and attribute declarations, as well as WSDL interfaces, operations, and faults.

The semantic annotations applied to the schema types help in web service discovery and potential service composition. The semantic annotations applied to WSDL interfaces and operations description provide categorization information necessary for future semantical based discovery (after publication of the service description in a registry).

The URIs appearing in the list assigned to the `modelReference` attribute are direct references to the ontological concepts. If direct references are not accessible, then

`liftingSchemaMapping` and `loweringSchemaMapping` attributes can provide references to data mapping transformation.

11.1.2. Schema Mapping

The `liftingSchemaMapping` and `loweringSchemaMapping` extension attributes apply in the XML Schema element declarations and type definitions. They serve as a place for declaration of the data mappings of XML Schema types to and from semantic models. The URIs from the list assigned as values to these attributes identify mappings.

The `liftingSchemaMapping` can be applied to XML Schema element declaration, `complexType` definitions and `simpleType` definitions. The specification does not define any rules on how a client processor should acquire details of the semantic model being referred to. However, the specification recommends that each URI should resolve to a document with a semantic model definition. In particular, a semantic model definition can reside within WSDL document.

NOTE:

The non-semantical service discovery can end up with the results which meet functional requirements of a client, but are not usable because of the mismatches in the the semantic model and the structure of the inputs and outputs of the discovered service's operations. For example, the client may operate on data including addresses build from street names concatenated with numbers, while the service may require all address parts separated. A lowering schema mapping applied to the client's data would turn them into an XML form acceptable by the service. The lifting schema mapping would invert the data acquired from a service back to the form acceptable by a client. In general, lifting schema mappings lift data from XML to a semantic model, whereas lowering schema mappings lower data from a semantic model into an XML structure.

11.2. Annotating WSDL Documents

Conceptually, the WSDL service descriptions consist of abstract and concrete parts. The SAWSDL specification focuses on annotating the abstract parts semantically.

In terms of the WSDL 2.0 component model, a model reference is a new property. In particular, when used on an element that represents a WSDL 2.0 Component, the `modelReference` extension attribute with a non-empty value introduces an optional property {model reference} whose value is a set of URIs taken from the value of this attribute. An empty model reference or no model reference are both reflected by the absence of the {model reference} property on a given component.

The mechanism for semantic annotation described so far can be also applied to the WSDL 1.1 based web service descriptions. All the XML attributes mentioned apply without modification. Annotation of XML Schema types in WSDL 1.1 with `modelReference`, `liftingSchemaMapping` or `loweringSchemaMapping` works in the same way as in WSDL 2.0. The `modelReference` annotation on element, complexType, simpleType and attribute defines the semantics of the input or output data of WSDL operations where these types are used.

However, semantic annotations in some cases apply to different elements then in WSDL 2.0 document structure. Moreover, to overcome a restrictions on attribute extension (and thus to facilitate operation annotations) a new annotation mechanism was introduced.

A WSDL 1.1 schema prohibits extension attributes on `operation` element. The problem was solved by introducing a general mechanism for adding extension attributes where attribute extensibility is not allowed, but element extensibility is allowed. This mechanism relies on adding a new extension element `attrExtensions` annotated with a `modelReference` attribute to the element being described. The `attrExtensions` SHOULD NOT be used where attribute extensibility is allowed. The `attrExtensions` element MUST NOT be used for SAWSDL annotations in WSDL 2.0. Attributes with the same namespace name and local name MUST NOT appear both on the `attrExtensions` element and on its parent element.

Thus, to annotate `operation` element one should add an `attrExtensions` child to it. The `modelReference` attribute of added element will specify the operation annotation.

A WSDL 1.1 `portType` corresponds to a WSDL 2.0 `interface` and is annotated in the same way. A `modelReference` applied provides a classification or other semantic descriptions of this element.

A `liftingSchemaMapping`, `loweringSchemaMapping` or `modelReference` attribute may be added to a `part` element, defined under a `message` element, to specify an input or output annotation that applies to the entire message part. Message parts are referenced from the `portType` structure in WSDL 1.1 that generally corresponds to the WSDL 2.0 `interface` structure.

In WSDL 1.1 fault is defined identically to input or output, i.e. as a `<fault>` subelement of the operation element. Annotation of the meaning of the fault needs to be done on the `<fault>` element in each operation where it occurs.

11.3. Sample of SAWSDL description

The example of WSDL 1.1 document presented in Section 10.3 can be enriched with the semantic annotation as follows:

```
<interface name="Informator" >
    <operation name="GetOrderPrice"
                 pattern="http://www.w3.org/2004/03/wsdl/in-out">
<sawsdl:attrExtensions
   sawsdl:modelReference="http://example.org/purchase#RequestOrderPrice">
<input messageLable = "In"
             element="skl:OrderPriceRequest"/>
      <output messageLable = "Out"
             element="skl:OrderPriceResponse"/>
    </operation>
  </interface>
```

11.4. SAWSDL API

WSDL documents can be processed programmatically with the aid of general (XML processing) or specific (SAWSDL processing) API. The list of specific API includes:

- SAWSDL4J which provides a clean object model for SAWSDL documents (available at `http://sawsdl4j.sourceforge.net`).
- Woden4SAWSDL which provides an implementation of WSDL 2.0 parser and API allowing SAWSDL parsing and creation (available at `http://lsdis.cs.uga.edu/projects/meteor-s/opensource/woden4sawsdl/index.html`).

- Radiant which is an eclipse plugin with graphical user interface for annotating existing WSDL documents into WSDL-S or SAWSDL via an OWL Ontology (available at `http://lsdis.cs.uga.edu/projects/meteor-s/downloads/index.php?page=1`).
- GRDDL with transformation for SAWSDL (available at `http://ns.inria.fr/grddl/sawsdl/`).

12. UDDI (Universal Description, Discovery and Integration)

UDDI (Universal Description, Discovery and Integration) is an open industry initiative running under the auspices of the OASIS (Organization for the Advancement of Structured Information Standards). The members of this initiative, involving the largest IT companies, considered the problem of describing, publishing and discovering information about services and service providers. The aim was to enable businesses to find each other in an easy, efficient and well-defined way, and to define the principles of businesses' interaction over the Internet. The conducted work resulted in a proposition of a global, platform-independent, open framework.

The framework is based on W3C (WorldWideWeb Consortium) and IETF (Internet Engineering Task Force) standards as XML, HTTP, DNS (Domain Name System) and benefits from SOAP and WSDL. The triple UDDI, SOAP and WSDL is recognized as a set of foundation standards of SOA (service-oriented architecture).

The framework includes a definition of a model and operations on a registry of Web services. The registry helps in realization of the concept of B2B (business-to-business) communication based on Web Services (see Figure 12.1). A typical scenario of the registry use is following:

- the provider publishes the service description specified in WSDL to the registry,
- the requester searches the registry for the information published,
- the requester sets up a connection with a service based on details received in the registry response (possibly creating its own service client in an automatic manner),
- both, provider and requester, use SOAP for communication with the registry.

Registry contains information on the companies and provided Web services. It is a kind of a "phone book" allowing searching by a type of business and type of service. The registry information scope is often described using the phone book metaphor: *The UDDI registry contains: green pages - a technical description of the service and its URL reference (by assumption the service described do not necessarily has to be a Web service); white pages - identification, addresses and other contact details of companies; yellow pages - a list of companies arranged by industrial classification.* However, this is only a metaphor. The real information model used in the registry is given in UDDI specification. Moreover, the information stored in the registry can have much broader application then only business description. It can be used, for example, to describe a hierarchy of an organization (department, unit, etc.) or information infrastructure (applications, servers, etc.)

The UDDI specification has evolved over time to reflect the providers and customers needs. All started with a version 1.0, released in 2000, which laid foundation for the registry of Internet-based business services. Version 2.0, released in 2001 and ratified as an OASIS Standard in 2003, aligned the specification with emerging Web services standards and provided a flexible service taxonomy. Next version, with a 3.0 number, released in 2004 and ratified as an OASIS Standard in 2005, supported secure interaction of private and public implementations as major element of service-oriented infrastructure. The current version is 3.0.2. For more information about UDDI, please refer to http://www.uddi.org/about.html.

IBM, Microsoft and SAP used to host public UDDI registry (called the UDDI Business Registry, UBR) a couple of years ago, but that was discontinued in the favour of private registries.

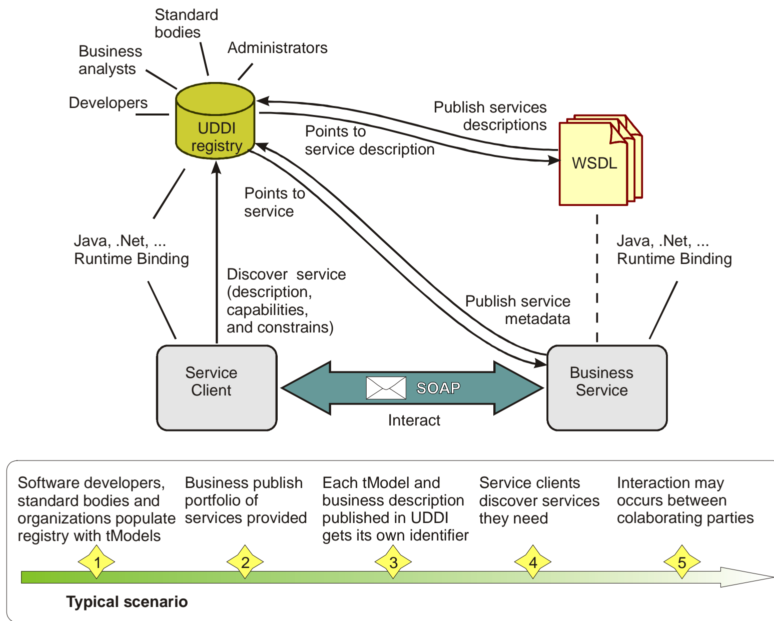


Figure 12.1. Typical UDDI registry use cases.

12.1. Technical Architecture

The UDDI technical architecture consists of three parts:

– the data model that allows creation and storage of information on businesses and web services

– the API specification that includes definitions of operations for finding and publishing data (and other operations) in management of rights,

– the registry that enables companies to advertise their business so other businesses can find potential partners.

The data model and the API specification have formal descriptions expressed in XML Schema language (see Table 12.1 and Table 12.2). The registry can be accessed traditionally (as a web application) or programmatically (as a web service), and can be implemented in various ways.

Table 12.1. UDDI Version 3.0.2 XML Schema files (available at `http://uddi.org/schema/`).

Description	XML Schema file
API Schema	uddi_v3.xsd
Custody Schema	uddi_v3custody.xsd
Subscription Schema	uddi_v3subscription.xsd
Subscription Listener Schema	uddi_v3subscriptionListener.xsd
Replication Schema	uddi_v3replication.xsd
Value Set Validation Schema	uddi_v3valueset.xsd
Value Set Caching	uddi_v3valuesetcaching.xsd
Policy	uddi_v3policy.xsd
Policy Instance Parameters	uddi_v3policy_instanceParms.xsd

Table 12.2. UDDI Version 3.0.2 WSDL Service Interface Descriptions files (available at `http://uddi.org/wsdl/`).

Description	WSDL file
API Binding	uddi_api_v3_binding.wsdl
API Port Type	uddi_api_v3_portType.wsdl
Custody Binding	uddi_custody_v3_binding.wsdl
Custody Port Type	uddi_custody_v3_portType.wsdl
Replication Binding	uddi_repl_v3_binding.wsdl
Replication Port Type	uddi_repl_v3_portType.wsdl
Subscription Binding	uddi_sub_v3_binding.wsdl
Subscription Port Type	uddi_sub_v3_portType.wsdl
Subscription Listener Binding	uddi_subr_v3_binding.wsdl
Subscription Listener Port Type	uddi_subr_v3_portType.wsdl
Value Set Validation Binding	uddi_vs_v3_binding.wsdl
Value Set Validation Port Type	uddi_vs_v3_portType.wsdl
Value Set Caching Binding	uddi_vscache_v3_binding.wsdl
Value Set Caching Port Type	uddi_vscache_v3_portType.wsdl

The UDDI registry is often called a cloud of services because it is logically centralized, but physically distributed, built from multiple UDDI nodes that synchronize their data by replication. Thanks to the replication, businesses can register themselves at any node of the UDDI registry, and the same information will be available in time at any other node of this registry (which possibly resides on different systems).

Multiple registries may form a group, known as an “affiliation” (see Figure 12.2). The aim of setting such a group is to permit controlled copying of core data structures between its members. To do this in a safe way an introduction of appropriate policies is required. Affiliated registries: share a common namespace for entity keys, have compatible policies for assigning keys to entities, have policies that permit publishers to assign keys.

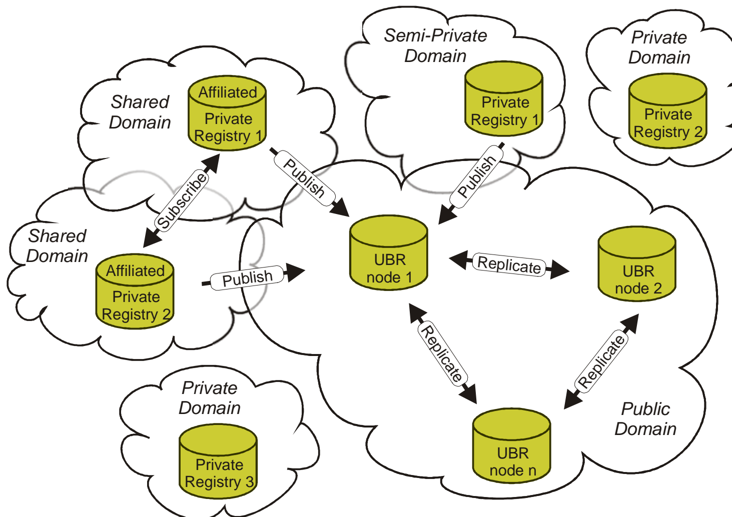


Figure 12.2. Affiliation of UDDI registries (inspired by: `http://www.uddi.org/pubs/uddi-tech-wp.pdf`).

UDDI nodes are servers that are compliant with the UDDI specification. They belong to the UDDI registry, and collectively manage a well-defined set of UDDI data. The UDDI registry, in turn, is comprised of one or more UDDI nodes.

The registry is itself a web service based on SOAP. There are some API sets defined for different operations on the registry. All operations are synchronous, and the UDDI specification provides details on them in a form of XML Schemas. Summarizing, the construction of UDDI registry conforms to the following rules:

- UDDI registry must have at least one node that offers a Web service compliant Inquiry API set.
- UDDI registry should have at least one node that offers a Web service compliant with the Publication, Security, and Custody and Ownership Transfer API sets.
- If a UDDI registry has multiple nodes, all nodes should offer Web services that are compliant with the Replication API set. Thanks to that, data supplied to one of the registry nodes can be replicated to all other nodes, so they will look identically, regardless the node they will be accessed from (replication occurs every 24 hours).
- The Subscription and Value Set API sets are optional for all nodes and all registries.
- A registry must make a policy decision for each policy decision point. It MAY choose to delegate policy decisions to nodes.

Some software tools and frameworks allow creating private UDDI registry, for example, BEA WebLogic Server, IBM WebSphere (commercial) and jUDDI (open source). However, the nodes of private registry do not synchronize with the nodes of other registries. Of course, this does not prohibit the synchronization of private registry nodes with one another.

12.2. UDDI data structures

UDDI information model is composed of instances (persistently stored in UDDI nodes) of data structures, called entities (formally described using XML Schema Language). The main entity types are following (described in API Schema, `uddi_v3.xsd`):

`businessEntity`: represents descriptive information about the business or provider and about the services it offers, such as: contact information, business category, business identifiers, the list of services provided. It includes one or more `businesService` structures containing service descriptions and technical information.

`businessService`: represents collection of related network services offered by a service provider defined by `businessEntity`. It includes information about particular service's binding, type, and category through one or more `bindingTemplates`.

`bindingTemplate`: represents technical information necessary to use a particular Web service. It contains references to `tModels`.

`tModel`: represents a *technical model* of reusable concept, such as a Web service type, a protocol used by Web services, or a category system. The use of `tModel` structure eliminates duplication of the same information in different places (e.g. in a situation where many services offer the same interface).

`publisherAssertion`: represents the relationship between `businessEntities` such as the manufacturer - supplier, contractor - subcontractor, etc.

The associations between the UDDI data structures are shown in Figure 12.3. Taking a closer look at it makes it clear that entities are organized into a hierarchical pattern. The business entities are at the top of the pyramid, they contain business services, and those

contain binding templates. Thus, to publish any information about services, at first businessEntity should be registered, then businessService, and then bindingTemplate.

Figure 12.3. The UDDI data structures and the relationships between them as defined in uddi_v3.xsd. There are some changes to the official UDDI Version 3.0.2. They affect, among the others, the bindingTemplate structure. The changes can be summarized as follows: a) removing the use of xsd:choice b) removing the use of the final attribute. The diagram hides attributes of elements (bindingTemplate has two optional attributes: serviceKey, bindingKey; businessEntity has one optional attribute businessKey; tModel has two optional attributes: deleted, tModelKey).

12.2.1. businessEntity

The businessEntity is the top-level data structure holding descriptive, possibly multilangual information about the business or organization.

The textual service descriptions are stored in businessService structures, and corresponding technical information are stored in bindingTemplate structures. These structures are associated with businessEntity through references.

Each businessEntity structure is uniquely identified by the value of its businessKey attribute. The value of this attribute, of the UUID type, MUST be omitted if the publisher wants the registry to generate a key during registration. The value of this attribute MUST be present, if a businessEntity is retrieved from a UDDI registry. The elements that a businessEntity might contain are following:

discoveryURLs – is a list of URLs that point to alternate, file based service discovery mechanisms;

name – is a name of a businessEntity (required, non-empty, can occur multiple times);

description – is textual information about the businessEntity (optional, can occur multiple times);

contacts – is a simple list of single contact information (optional, can occur multiple times);

businessServices – is a list of business services provided by a businessEntity;

identifierBag – contains a list of identifiers used for other purposes than a businessKey, valid in their own identifier systems (like tax identifier);

categoryBag – contains a list of business categories which a businessEntity can be associated with (like industry, product category or geographic region);

Signature – contains a digital signature created in accordance with the XML-Signature specification (optional, but MUST be provided if a businessEntity was digitaly signed).

12.2.2. businessService

A businessService structure represents a group of web services and includes simple, textual information outlining the purpose of individual Web services. It contains: names, descriptions and classification information, potentially in multiple languages. The technical information is stored in the bindingTemplate structures provided in the contained bindingTemplates list.

A businessService has two attributes: serviceKey and businessKey. The serviceKey attribute uniquely identifies a given businessService in all UDDI registries. The value of serviceKey follows the same rules as the value of businessKey attribute of businessEntity (it MUST be omitted when registering an entity and MUST be present when retrieving an entity). The businessKey attribute represents logical parent-child relationship between a businessEntity and a businessService. Every businessService has exactly one businessEntity parent, uniquely identified by businessKey attribute. However, the value of a businessKey attribute of a businessService may differ from the value of a businessKey attribute of a parental businessEntity. When it happens, it indicates a service projection.

A service projection allows a business or organization to include in its businessEntity a businessService offered by some other business or organization. A projected businessService is made part of a businessEntity by a reference as opposed to by containment. Projections to the same service can be made in any number of business entities.

The elements that a businessService might contain are following:

name – is a name of a businessEntity (required except when indicating a service projection, non-empty, can occur multiple times);

description – is textual information about the businessService (optional, can occur multiple times);

bindingTemplates – is a list of technical descriptions for theWeb services provided;

categoryBag – contains a list of business categories which a businessService can be associated with (like industry, product category or geographic region);

Signature – contains a digital signature created in accordance with the XML-Signature specification (optional, but MUST be provided if a businessService was digitaly signed).

12.2.3. bindingTemplate

A bindingTemplate structure represents technical information needed by applications to bind and interact with services described in the businessService structure. The bindingTemplate includes information on service protocol binding, access points (given with URL) or an indirection mechanism leading to the access point, a notice whether service is independent, etc. The type of Web service being offered is provided using references to tModels, application-specific parameters, and settings. Because the same service can be implemented in many ways, it can have various bindingTemplate entities associated with it, with different set of protocols and different network addresses.

A bindingTemplate has two attributes: serviceKey and bindingKey. The bingingKey attribute uniquely identifies a bindingTemplate in all UDDI registries. The value of bindingKey follows the same rules as the value of businessKey attribute of businessEntity (it must be omitted when registering an entity and must be present when retrieving an entity).

The serviceKey attribute uniquely identifies the businessService that contains the bindingTemplate (each bindingTemplate is the child of a single businessService which is referenced by a serviceKey). The value of serviceKey follows simmilar rules as the value of bindingKey attribute (it may be ommited when registering a bindingTemplate entity and this entity is a part of a fully expressed businessService element, and must be present when retrieving a bindingTemplate).

The elements that a bindingTemplate might contain are following:

description – is textual information about the bindingTemplate (optional, can occur multiple times, potentially in multiple languages);

accessPoint – is a string used to convey the network address suitable for invoking the Web service being described (typically an URL, but may be any other locator, as an e-mail address, a telephone number, etc.);

hostingRedirector – is a deprecated element, functionality of which is now covered by the accessPoint (is mutually exclusive with accessPoint);

tModelInstanceDetails – is a structure containing a list of one or more tModelInstanceInfo elements, each with tModelKey attribute. All they form a kind of stamp that can be used to identify compatible services;

categoryBag – is a container for categories which a bindingTemplate can be associated with (as, for example, „test" or „production").

Signature – contains a digital signature created in accordance with the XML-Signature specification (optional, but must be provided if a bindingTemplate was digitally signed).

12.2.4. tModel

Technical Model, or tModel for short, is used to describe technical aspects, important for the developers, of a variety of business, service, and template entities registered in UDDI registry. However, the technical descriptions are not registered in the UDDI registry. Instead, tModel provides the addresses to the documents and supporting documentation (WSDL,

XSD, and other documents that outline and specify the contract and behavior) along with metadata.

tModel can be used to register any unique concept or construct within UDDI registry. Each registered tModel entity has unique key identifying it within registry. The use of such keys prevents duplication of the same information in different places, as it would happen in a case of several services offering the same interface. Thus, UDDI data structures can refer to a particular registered tModel using its key. The UDDI specification defines a set of common tModels that can be used canonically to model information in registry in this way.

A tModel has two attributes: deleted and tModelKey. The tModelKey attribute uniquely identifies a tModel entity in all UDDI registries. The value of tModelKey follows the same rules as the value of businessKey attribute of businessEntity (it MUST be omitted when registering an entity and MUST be present when retrieving an entity). The deleted attribute appears in the retrieved tModel data as an information-only field. It indicates whether tModel was deleted from a registry or not. Two allowed values for this attribute are "true" and "false".

The elements that a tModel might contain are following:

name – is a name of a tModel (required, non-empty, can occur multiple times, SHOULD be formatted as a URI, and the xml:lang attribute SHOULD NOT be used);

description – is short, textual information about the tModel (optional, can occur multiple times, potentially in multiple languages);

overviewDoc – contains an URL reference to remote descriptive information or instructions related to the tModel (optional, can occur multiple times). More exactly, overviewURL element, included in overviewDoc, holds an URL. An overviewURL has an optional useType attribute with a value indicating the type of referenced document ("text" value indicates that the overviewURL refers to additional textual information, and "wsdlInterface" value indicates that the overviewURL refers to a WSDL interface document).

identifierBag – contains a list of identifiers used for other purposes than a tModelKey, valid in their own identifier systems.

categoryBag – is a container for categories which a tModel can be associated with (showing, for example, its technical type).

Signature – contains a digital signature created in accordance with the XML-Signature specification (optional, but MUST be provided if a tModel entity was digitally signed).

12.2.5. publisherAssertion

A publisherAssertion structure links two or more business entities represented by businessEntity structure. It is used in a case when there are some associations, such as the manufacturer - supplier, contractor - subcontractor, etc. between these entities. To make these associations visible in the registry (for customers who wish to find information about cooperating businesses), these structures should be registered by the both entities involved. publisherAssertion structure consists of three elements: fromKey (a key of the first entity), toKey (a key of the associated entity) and keyedReference (a reference that defines the type of association in terms of pairs: KeyName, keyValue inside a tModel referenced by tModelKey).

12.2.6. operationalInfo

An `operationalInfo` structure stores information about structures published in a registry, including: date and time of structure creation and modification, ID of the UDDI node where publication occurred, and ID of the publisher.

12.3. UDDI Interfaces

UDDI registry can be accessed programmatically for the purposes of manipulating or using data stored within it. The UDDI specification standardizes behaviour and communication with and between implementations of UDDI registries. XML Schemas support formal definitions of UDDI APIs and basic data types used in all flows of information. Thus, the operations of the API sets correspond to messages of well defined structure and underlying UDDI datatypes.

The UDDI API's are grouped into API sets as described below. The most commonly used Node API Sets are: UDDI Inquiry, UDDI Publication and UDDI Security.

Node API Sets:

- UDDI Inquiry – contains operations for querying the registry for details on registered entities
- UDDI Publication – contains operations for publishing entities into the registry
- UDDI Security – contains operations for authentication handling
- UDDI Custody Transfer – contains operations for transferring ownership and custody of entities
- UDDI Subscription – contains operations for retrieving information on entities in a timely manner using a subscription format
- UDDI Replication – contains operations related to data replication between registry nodes

Client API Sets:

- UDDI Subscription Listener – contains operations for receiving subscription results
- UDDI Value Set – contains operations related to keyed reference values validation

12.3.1. Inquiry API Set

Inquiry interface is used primarily to search for service descriptions in the registry (with the use of different classification schemes). It contains operations that can be organized in two groups: browsing operations and drilling-down operations. Browsing operations (`find_xxx`) are used to get a coarse view and check the contents of a large portion of information. Drilling-down operations (`get_xxx`) are used to search for specific, detailed information based on coarse information.

A typical use case scenario starts with a browsing request issued usually by the registry client from inside of dedicated software. The results of browsing (keys) are used in drilling-down requests that provides detailed information contained in the underlying data structures (such as `businessEntity`, `businessService`, `bindingTemplate` and `tModel`). After receiving these details, the client application can call the service found.

To search requests are:

`find_binding` – returns a list of network services bingings matching the criteria specified in the input arguments (on technical information);

`find_business` – returns a list of business entities matching the search criteria;

`find_relatedBusinesses` – discovers related business compounds.

`find_service` – returns a list of network services matching the search criteria;

`find_tModel` – returns a list of `tModel` structures matching the search criteria;

The demands of drilling include:

`get_bindingDetail` – returns the complete information about the service `bindingTemplate` structure for all values of `bindingKey` listed in the request;

`get_businessDetail`– returns the information in the `businessEntity` structure for all business entities whose values are `bindingKey` listed in the request ;

`get_businessDetailExt` – returns extended information on a business entity in a `businessDetailExt` structure;

`get_operationalInfo` – returns the information in the `operationalInfos` structure (containing data such as time and date of creation of a structure, the date and time Last modified, node identifier, the entity which has been published, the service provider identifier) for all entities whose values are `entityKey` listed in the request.

`get_serviceDetail` – returns the details in the structure of `businessService` for services for which the value of `serviceKey` listed in the request;

`get_tModelDetail` – returns the information for `tModel` entities identified by the values of `tModelKey` listed in the request.

12.3.2. Publication API Set

The interface allows modifying registry content by registering a new, and updating or removing an old information. The interface contains fourteen operations:

`add_publisherAssertions` – adds one or more relationship assertions to the existing assertion collection;

`delete_binding` – removes an existing `bindingTemplate` from a `bindingTemplates` collection that is part of a specified `businessService` structure;

`delete_business` – removes the registered `businessEntity` information from the registry;

`delete_publisherAssertions` – removes one or more assertion from the assertion set managed by a particular publisher account.

`delete_service` – removes the existing `businessService` from the `businessServices` collection that is part of a specified `businessEntity`;

`delete_tModel` – hides registered information about a `tModel` (models can not be deleted normally, except by administrative petition);

`get_assertionStatusReport` – provides administrative support for determining the status of all assertions made involving any `businessEntity` controlled by the requesting publisher account. The status is included in the `assertionStatusReport` returned.

`get_publisherAssertions` – gets a list of all relationship assertions associated with a specific publisher account in `publisherAssertions` structure;

`get_registeredInfo` – returns an abbreviated synopsis of all information currently managed by a given individual;

`save_binding` – creates or updates `bindingTemplate` information;

`save_business` – creates or updates `businessEntity` information;

save_service – creates or updates complete information about a businessService exposed by a specified businessEntity.

save_tModel – creates or updates complete information about a tModel;

set_publisherAssertions – manages all relationship assertions for an individual publisher account (replaces any existing assertions, and causes any old assertions that are not reasserted to be removed from the registry).

12.3.3. Security Policy API Set

This interface contains two methods used to draw and release a token required to perform safe operations:

discard_authToken – is used to inform a node that an authentication token it has obtained is no longer required and should be considered as invalid (equivalent to logout from the system).

get_authToken – is used to request an authentication token potentially required for performing operations included in the Inquiry API Set, Publication API Set, Custody and Ownership Transfer API Set, and Subscription API Set. The requested token issued from a UDDI node has the form of an authInfo element (equivalent to login to the system).

12.3.4. Custody and Ownership Transfer API Set

The operations of this interface enables any nodes of a registry to cooperatively transfer custody of one or more businessEntity or tModel structures from one node to another, as well as allowing the transfer of ownership of these structures from one publisher to another. Associated entities of a businessEntity such as its businessService, bindingTemplate, and publisherAssertion structures are transferred as part of the custody transfer of the business entity.

A publisher keeps an ownership on the entity it has created (a publisher is the owner of the entity). A custodial node must maintain a relationship of ownership between an entity and its publisher by means of authorization mechanisms. Every node of a multi-node registry must guarantee the integrity of an entity's custody. As such, a node must not permit changes to an entity unless it has custody of it.

The base data structure used in this API Set is transferToken. This structure represents the one-time authority to transfer ownership of a specific set of entities to any publisher and to transfer custody of them to any node in the registry. The authority represented by a transferToken expires after some period of time, per node policy.

The list of operations of this inteface is following:

discard_transferToken – is used to discard a transferToken obtained through the get_transferToken API at the same node;

get_transferToken – is used to initiate the transfer of custody of one or more businessEntity or tModel entities from one node to another;

transfer_entities – is used by publishers to whom custody is being transferred to actually perform the transfer. The recipient publisher must have an unexpired transferToken that was issued by the custodial node for the entities being transferred;

transfer_custody – is used by the custodial node to ensure that permission has been granted to transfer custody of the entities that the target publisher has requested (is invoked by the target node in response to transfer_entities). The transfer_custody API is

in the replication namespace since it is sent from one node to another node in a registry using replication.

12.3.5. Subscription API Set

Operations of the interface are used to monitor activities in the registry. They are designed so flexibly that it is possible to monitor new, changed and deleted entries for each of the following entities: `businessEntity`, `businessService`, `bindingTemplate`, `tModel` and entities related through `publisherAssertion`. Operations may be invoked in a synchronous or asynchronous mode (asynchronous mode relies on sending notifications to the subscriber by UDDI node). Operations of subscription interface are following:

`save_subscription` – creates a new subscription, changes an existing subscription or renews an existing subscription;

`delete_subscription` – cancels one or more subscription;

`get_subscriptions` – returns a list of existing subscriptions previously saved by the applicant;

`get_subscriptionResults` – returns the registry data pertaining to the specific subscription within a specified time;

`notify_subscriptionListener` – optional method implemented by a subscriber and pre-subscribed in UDDI, called by the UDDI node in order to notify subscriber about data changes (about new, modified or deleted data matching the subscription).

12.3.6. Value Set API Set

The "value sets" term stands for all classification and identification systems used in UDDI. Value sets may be "checked" (having the use of them checked to recognize whether it conforms to the formal requirements of the value set) or "unchecked" (do not having their use checked). UDDI provides the opportunity to register value sets and control the validation process to third parties. Caching collections of such external values can be also supported (with external validation). Publicly available interfaces to support external validation set of operations include:

`validate_values` – used by nodes to allow external network service providers to validate value sets (to determine whether `keyedReferences` or `keyedReferenceGroups` are correct). Returns a `dispositionReport` structure;

`get_allValidValues` – used by nodes to obtain the set of valid values from cacheable checked value sets (applies to nodes supporting caching of valid values). Returns an empty message or a `dispositionReport` structure.

12.4. Using WSDL Definitions with UDDI

As described in chapter 12 UDDI registry stores metadata of Web service and references to the specifications on implementation. According to the OASIS UDDI Technical Note *Using WSDL in a UDDI Registry*, Version 2.0.2 (available at `http://www.oasis-open.org/committees/uddi-spec/doc/tn/uddi-spec-tc-tn-wsdl-v2.htm`) the information from WSDL documents can be referenced in UDDI `businesServices` (service documents) and `tModels` (binding documents). The corresponding mapping is presented in Figure 12.4.

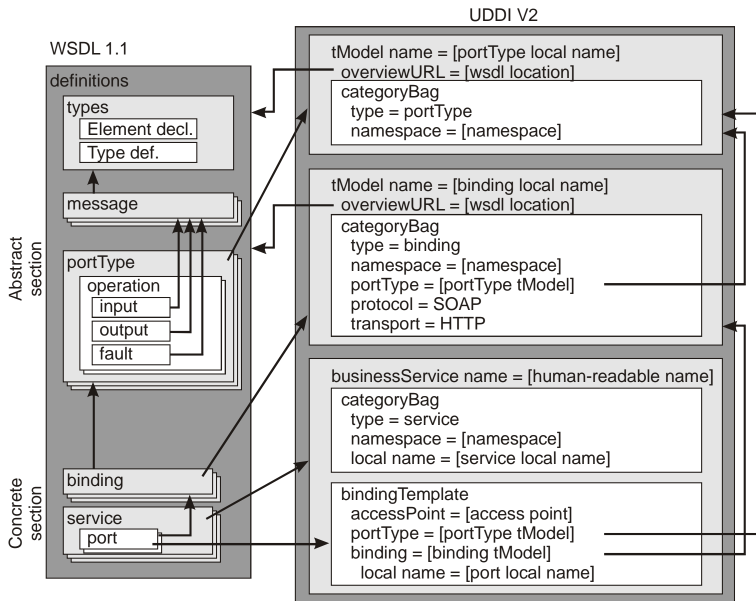


Figure 12.4. Methodology for mapping WSDL 1.1 definitions to the UDDI V2 and UDDI V3 data models (based on UDDI Technical Note *Using WSDL in a UDDI Registry*, Version 2.0.2).

The example below shows `<tModel>` definition with a reference to the WSDL document with URL of this document provided inside overviewURL.

```
<tModel authorizedName="..." operator="..." tModelKey="...">
  <name>CustomService</name>
  <description xml:lang="en">
WSDL description of the CustomService
  </description>
  <overviewDoc>
<description xml:lang="en">WSDL source document</description>
<overviewURL>http://example.org/wsdl/CustomService.wsdl</overviewURL>
  </overviewDoc>
  <categoryBag>
<keyedReference tModelKey="uuid:..."
keyName="uddi-org:types" keyValue="wsdlSpec"/>
  </categoryBag>
</tModel>
```

13. WS-CDL (Web Services Choreography Description Language)

The WS-CDL is a W3C Candidate Recommendation for a choreography language based on XML. The language itself includes constructs and elements for specifying common and observable behaviours of collaborating parties that need to interact in order to achieve some goals. The language allows describing rules of message exchange for multiple web service-based participants. Some of its aspects are inspired by the pi-calculus. The long-living and stateful interactions are defined from a global point of view, independently of how these behaviours are implemented internally. Thus, the description provided is presented in a global and neutral perspective rather than from a perspective of any particular participant.

The WS-CDL specification was initially designed by Oracle. In September 2003 it was submitted into the W3C Web Services Choreography Working Group. The first Working Draft was published in April 2004. The WS-CDL specification as W3C Candidate Recommendation was published on 9th November 2005. Since then the specification did not mature enough to become a Recommendation. It has not been implemented by the main software vendors and has not got their support. However, there are some modelling tools that support it. The example is pi4soa project, which is described on the following web page: `http://sourceforge.net/apps/trac/pi4soa/wiki`.

The W3C Web Services Choreography Working Group was closed on the 10th July 2009. The three working drafts related to the WS-CDL: Primer [47], published on 19th June 2006, Requirements [49], published on 11th March 2004, and Model overview [48] published on 24th March 2004, got a retired status.

The list of standards related to BPM and Choreography with a short description is provided in Table 13.1.

Table 13.1. Standards related to BPM and Choreography with short description.

Standard	Organization	Type
Business Process Execution Language (BPEL)	OASIS	Execution Language. BPM's most popular language; represents a process as XML with web services bindings
Business Process Modelling Notation (BPMN)	BPMI	Notation language. Graphical language with a mapping to BPEL
Business Process Modelling Language (BPML)	BPMI	Execution language. An XML process language similar to BPEL
Business Process Query Language (BPQL)	BPMI	Administration and monitoring interface. Management interface to a business process management infrastructure that includes a process execution facility (process server) and a process deployment facility (process repository).
Business Process Semantic Model (BPSM)	BPMI	Process metamodel, in fashion of Object Management Group (OMG) Model-Driven Architecture (MDA)
Business Process Extension Layer (BPXL)	BPMI	BPEL extension for transactions, human workflow, business rules
UML Activity Diagrams	OMG	Notation language
Workflow Reference Model	WfMC	A basic architectural approach to workflow/BPM
XML Process Definition Language (XPDL)	WfMC	Execution language. An XML process language similar to BPEL
Workflow API (WAPI)	WfMC	Administration and monitoring, human interaction, system interaction. A functional and administrative API with definitions in C, IDL, and COM
Workflow XML (WfXML)	WfMC	An XML language for web service-based communication between workflow runtime engines (similar to choreography).
Business Process Definition	OMG	Execution language and/or notation language, as MDA

Metamodel (BPDM)		metamodel. A model for a BPM process language constructed using the Model Driven Architecture (MDA)
Business Process Runtime Interface (BPRI)	OMG	Administration and monitoring, human interaction, system interaction, as MDA metamodel, an MDA model for a functional and administrative BPM API
Web Services Choreography Interface (WSCI)	W3C	Choreography, a mature XML language for web services choreography, or the stateful, process-oriented interactions of web services among multiple participants
Web Services Choreography Description Language (WS-CDL)	W3C	Choreography. The W3C's official XML choreography language
Web Services Conversation Language (WSCL)	W3C	Choreography. A basic but elegant XML choreography language
XLANG	Microsoft	Execution language. An early XML process language;XLANG influenced the design of BPEL
Web Services Flow Language (WSFL)	IBM	Execution language. An early XML process language, which also influenced the design of BPEL
Business Process Schema Specification (BPSS)	OASIS	Choreography (and collaboration). A process language for business-to-business collaboration

13.1. Different views on business processes modelling

In the software and system engineering domain business process modelling and business process management play a very important role. They provide the bases for designing and implementing enterprise information system. Both terms share the same abbreviation, BPM, although their meaning is slightly different.

In general, a **business process modelling** is a method of improving organisational efficiency and quality. It focuses on designing and analysing a process model (also abbreviated to BPM) reflecting a flow of activities in a particular business or organisational unit. The model may have a structural representation, can be provided in a form of a description or can be visualized graphically in a diagram.

A **business process management** attempts to improve business processes continuously. This is a kind of activity or methodology oriented at improving effectiveness and efficiency while striving for innovation, flexibility, and integration with technology. It could therefore be described as a “process optimization process.”

A **business process** is a collection of related, structured activities or tasks, performed by their relevant roles, to produce a specific service or product for a particular customer or customers, and to purposefully achieve the common business goal.

The common understanding of all business processes and their relationships among participants involved is essential when implementing enterprise information systems. This common understanding has a different flavour in different contexts. Regarding coordination or control of individual web services in the scope of one, coherent overall process, there are two concepts related: “orchestration” and “choreography”. Introducing them shortly: orchestration refers to coordination of a single participant's process from a local, subjective level, whereas choreography refers to collaboration of multiple participants from a global view.

In Figure 13.1 the business processes of Buyer, Seller and CreditAgency are presented (for more details refer to the last section of this chapter). It can be seen that the exchange between these participants relies on sending messages with requests and responses. Thus, from the external perspective, the state of the collaboration can be determined only through the observation of messages being exchanged (see Figure 13.2).

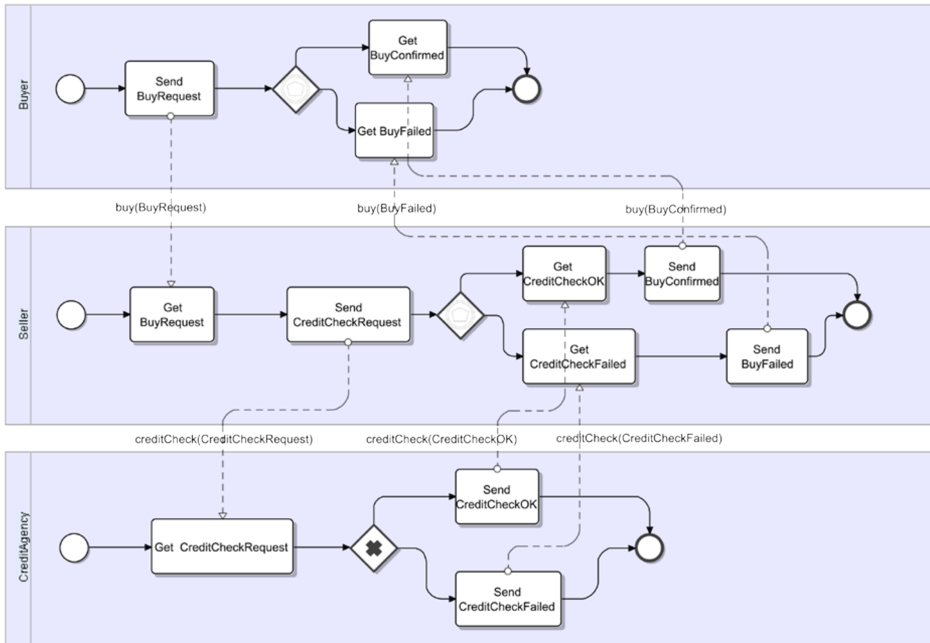


Figure 13.1. The example of processes in Buyer-Seller-CreditAgency cooperation (resources used: http://pi4soa.sourceforge.net).

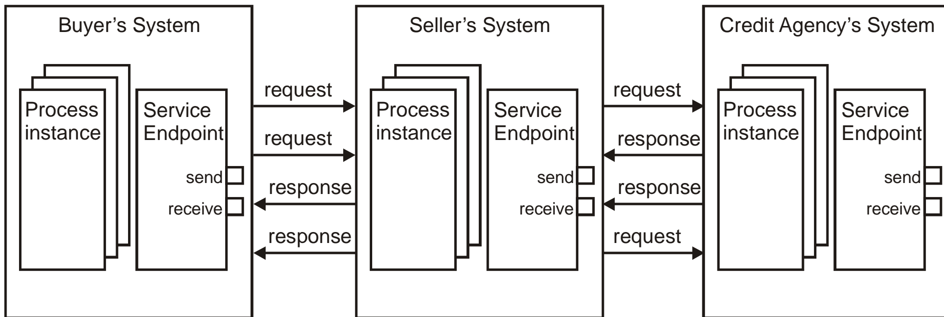


Figure 13.2. The Buyer-Seller-CreditAgency choreography.

13.1.1. Orchestration

Generally speaking, an orchestration is all about coordination of web services within a single process from a local, subjective perspective. An orchestration model of a particular service defines both the communication actions and the internal actions required for this service delivery. The communication actions involve externally and/or internally visible message exchanges and dependencies between them. The externally visible message exchanges appear while executing operations exposed to external parties through the service's behavioural interface(s). The internally visible message exchanges are parts of the process execution that are not exposed to external parties as they do not have to know any details about them. The internal actions may include processing which is not exposed directly as a service (data processing or running dedicated applications locally). Moreover, such actions

can be responsible for recording externally and internally visible states, essential for a proper business logic execution.

An orchestration can be expressed using an execution language, such as BPEL. So, once defined, it can be executed by an orchestration engine automatically and exposed as a service that can be invoked through an API. However, orchestration does not describe a coordinated set of interactions between two or more parties. What orchestration, and therefore BPEL, does is to enable a user to define a new service from existing services and present the result as a service which is a kind of recursive composition. But all this is done from a single controller perspective.

```
Orchestration = Executable Process
```

13.1.2. Choreography

Choreography is about global coordination of several services. It focuses on how to build stateful, conversational, long-running, multiparticipant processes out of basic stateless, atomic web service operations. It concerns collaboration of two or more participants, aimed at achieving a common goal.

A choreography model describes externally observable interactions that exists between collaborating participants within a global, participant agnostic perspective, and defines the common observable behaviour. It sets some jointly agreed, information driven reactive rules that control the observable information exchanges. The interactions between collaborating participants are captured with potential information alignment and dependencies, as, for example, control-flow dependencies, data-flow dependencies, transactional dependencies, message correlations, time constraints.

Choreography does not describe any internal action that occurs within a participating service such as an internal computation or data transformation. They are not considered if they are not visible externally.

Choreography is not an executable process. It serves rather as a contract between parties that clarifies all details of their collaboration. This contract can be written in choreography language as WS-CDL. The WS-CDL language allows declaration of multi-party contracts and is somewhat like an extension of WSDL: WSDL describes web service interfaces, WS-CDL describes collaborations between web services.

```
Choreography = Multi-party Collaboration
```

13.2. WS-CDL document

A WS-CDL document is simply a set of definitions specified with the use of some obligatory and optional XML constructs, following the syntax defined in `ws-cdl.xsd` schema (see Figure 13.3). A WS-CDL document includes choreography description with precise definitions of interactions between collaborating parties written using XML syntax with limited set of used terms. The specification of WS-CDL language permits extensibility elements and/or attributes being declared inside any of WS-CDL language elements. However, when used, they have to be defined in an XML namespaces, different from that of WS-CDL. Moreover, they must not contradict the semantics of any of elements or attributes from the WS-CDL namespace. In a WS-CDL document additional information can be also embedded. The container for it is a `<description>` element. This element can be nested in any other WS-CDL language element. The contents of `<description>` may refer to some semantic definitions and provide explanation of meaning and use of a nesting element. This information can be provided in multiple different human readable languages and in different

forms. However, the WS-CDL parsers are not required to parse nor interpret a `<description>`'s content. The acceptable forms of provided additional information can be following:

- plain text, HTML or other non-encoded text formats (corresponding to `text/plain`, `text/html`, `text/sgml`, `text/xml` and other MIME types),
- a reference to a document with a wider description (it may contain an URI to it),
- a machine readable description (it may be expressed in one of the semantic description languages as RDF or OWL and may contain an URI).

The choreography is about collaboration of multiple parties and their interactions through information exchange. So the main concepts in WS-CDL specification are: choreography, interaction, and channel. WS-CDL defines rules on passing information around channels (web service-based communication links) in the spirit of the pi-calculus. Similar to other XML applications, WS-CDL provides the syntax for element declaration and referencing. It relies on assigning matching values of `name` and `typeRef` attributes of two different elements, where `name` is treated as an element identifier and `typeRef` as a reference to it. It is important to understand that in WS-CDL document two elements can starts with the same XML tag name and have different syntax and semantics (one is a declaration, the other is a reference, as, for example, `roleType` in `package` and `roleType` in `participantType`).

Within choreography, information is always exchanged between participants within or across trust boundaries. The interactions occur between roles exhibited by participants, but only those which take part in a declared relationship. Thus, a choreography description in WS-CDL includes at minimum:

- a set of roles that represent a service with named behaviour which can be provided in a form of WSDL description,
- relationships between those roles,
- channels used by roles to interact, and
- a choreography block that uses channels to describe interaction.

Using this basic set of constructs one can describe typed and unambiguous service connections that enable collaboration. However, more complex collaboration requires more detailed description. A powerful feature of WS-CDL are guarded interactions – interactions that may occur only if some declared pre or post conditions are met. It is also possible to model interaction branching by introducing non-observable conditionals for observable exchanges. This can be modelled using existential predicate returning true when a certain pattern of observable interaction appeared. Moreover, the interactions (and choreographies already declared) can be combined into sequences, parallel activities and so on, with further rules achieved through their structural composition.

In a WS-CDL document a `package` is a root element. It is a container for such elements as `informationType`, `token`, `tokenLocator`, `roleType`, `relationshipType`, `participantType`, `chanelType`, and `choreography` (including `activity` declarations). The `roleType`, `relationshipType`, `participantType`, and `channelType` elements define collaborating participants and their coupling. A `roleType` enumerates potential observable behavior that `participantType` can exhibit in order to interact. A `relationshipType` identifies the mutual commitments that must be made for collaborations to be successful. A `participantType` groups together those parts of the observable behaviour that must be implemented by the same logical entity or abstract organization. Thus, a participant is abstractive modelled by a `participantType`, a role by

a roleType, and a relationship by a relationshipType (they are used when describing choreography).

At a certain level of collaboration declared roles must share some knowledge about states they entered what is essential for the coordination of their actions. This knowledge should be available globally or, at least, should be accessible for interacting roles. In practice this problem is solved by introducing a coordination mechanism based on use of variables. A variable may contain information about commonly observable objects, such as information exchanged or interactions performed between roles involved. For coordination purpose only a part of observed information may be sufficient. Such piece of data can be aliased by a token and extracted using tokenLocator. In WS-CDL document any variable of exchange variables, state capturing variables and tokens, holds data of informationType.

13.3. package

The <package> is a root element of any WS-CDL document. It aggregates a set of elements defining several types and choreography under a common namespace. The types are defined in informationType, token, tokenLocator, roleType, relationshipType, participantType and channelType elements. These definitions may be used in all choreographies defined within a package, and inside individual choreography type definitions. They are similar in some sense to the types in XSD. Once declared, these types can be used everywhere they are needed in a valid scope. The syntax of the <package> is following:

```
<package
   name="NCName"
   author="xsd:string"?
   version="xsd:string"?
   targetNamespace="uri"
   xmlns="http://www.w3.org/2005/10/cdl">

   <informationType/>*
   <token/>*
   <tokenLocator/>*
   <roleType/>*
   <relationshipType/>*
   <participantType/>*
   <channelType/>*

   Choreography-Notation*
</package>
```

The attributes name, author, and version store authoring properties of the document. The targetNamespace attribute defines the namespace for all WS-CDL type declarations. Other types can be included using an inclusion mechanism as in the example below:

```
<xi:include href="someGenericVariableDefinitions.xml"/>
<xi:include href="externalChoreography.xml"
xpointer="xpointer(//choreography/variable[1])"/>
```

Included declarations may be associated with different namespaces.

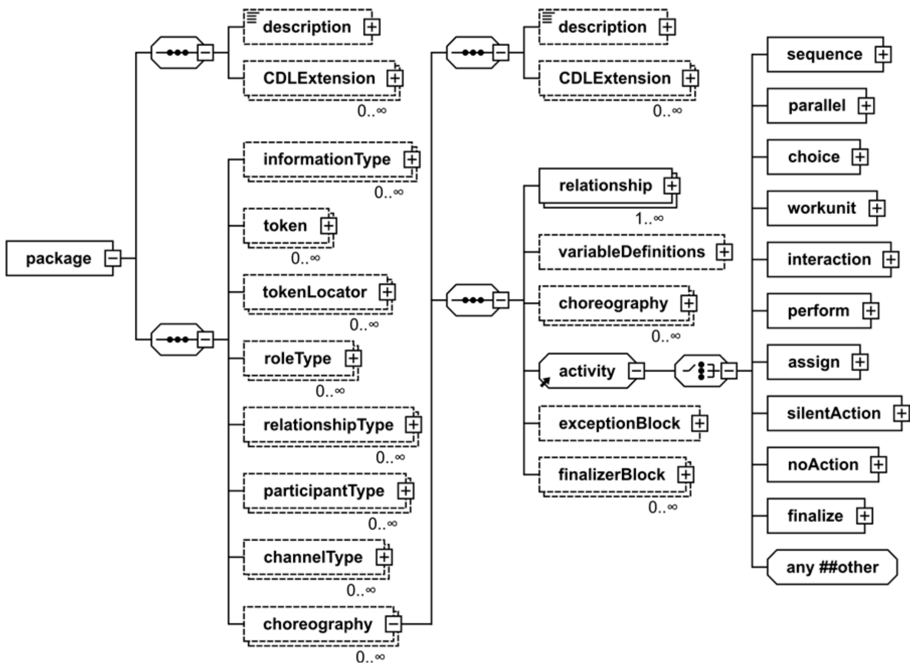


Figure 13.3. Diagram showing the content of the `<package>` element (with some nested elements collapsed and without elements' attributes, as defined in the current WS-CDL specification).

13.3.1. informationType

An `<informationType>` element is a place for declarations of abstract data types and elements used within choreography, as, for example, variables (storing messages or part of a messages or states resulting from evaluation of some expressions) or tokens. The syntax of the `informationType` is following:

```
<informationType name="NCName"
                 type="QName"?|element="QName"? />
```

where `name` is an obligatory attribute, unique within a choreography package, and used as a name of declared type, and `type` and `element` are optional, mutually exclusive attributes. The `type` is used as a reference to WSDL 1.1 message type or an XML Schema simple type, while `element` as a reference to a WSDL 2.0 Schema element or an XML Schema element (see Figure 13.4).

13.3.2. token, tokenLocator

Sometimes a choreography definition demands some complex types that have parts of the same type. In a WS-CDL it is possible to refer to such parts in a consistent, systematic manner. This can be done with the aid of `<token>` and `<tokenLocator>` element declarations.

The `token` represents an alias for a part of information. This part can be extracted from a complex type using XPath expression declared in `tokenLocator` associated with this `token`. One `tokenLocator` would be required for each combination of `token` and complex type that contains the token field. The use of `token` and `tokenLocator` is especially productive in choreography synchronization. Because collaborating participants do not share

memory, the only way for information sharing is to exchange it or derive it from the observations. In both cases, the type of information analysed is usually complex.

The syntax of the `token` is following:

```
<token  name="NCName"  informationType="QName" />
```

where attributes `name` and `informationType` are used, respectively, to specify a distinct name for the token, and to identify the type of a piece of information the token is an alias of.

The syntax of the `tokenLocator` is following:

```
<tokenLocator  tokenName="QName"
               informationType="QName"
               part="NCName"?
               query="XPath-expression" />
```

where attributes `tokenName` and `informationType` are used, respectively, to specify the name of the `token` that the `tokenLocator` is associated with, and to identify the type of the document on which the `query` is performed to locate the `token`. The values assigned to these two attributes are simply references (names of declared elements). The `part` attribute is optional. It defines a document's part that should be a subject of the query. It can be used to define a querable part of a multipart message described in WSDL1.1, but cannot be used for WSDL 2.0 based descriptions. Summarizing: `tokenType` specifies name and type as an alias to a piece of information within a document, `tokenLocator` specifies rules for selecting a piece of information within a document (see Figure 13.4).

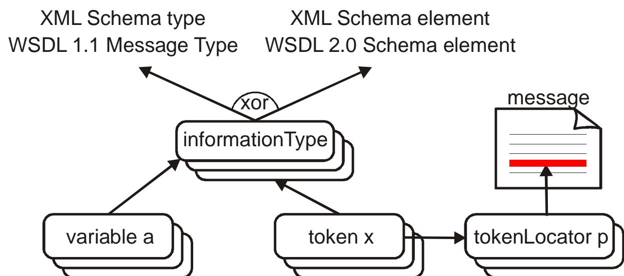


Figure 13.4. Dependencies between and use of `informationType`, `token` and `tokenLocator` elements.

13.3.3. roleType

A `<roleType>` element defines an observable behaviour a participant can exhibits in order to interact with other parties. This element must contain at least one `<behavior>` element that specifies the operations supported by this `roleType`. The syntax of a `roleType` is following:

```
<roleType name="NCName">
   <behavior name="NCName" interface="QName"? />+
</roleType>
```

where `name` attributes in `roleType` and `behavior` are unique names of corresponding elements. The name of a `roleType` can be referenced later from within `relationshipType` and `channelType` elements. The optional `interface` attribute of nested `behavior` element refers to a WSDL reference. A `behavior` without an `interface` attribute describes a behavior of a `roleType` without requirement to support any specific interface.

In the WS-CDL specification a given roleType cannot be associated with more than one participantType. However, the pi4soa tools include an extension to the standard, making it possible to treat roleType in the same manner as a Java interface, and have more than one participant, i.e. equivalent to a Java class, implementing the same roleType. Thanks to that, interactions defined within sub-choreographies can be reused in conjunction with multiple participant types.

13.3.4. relationshipType

The <relationshipType> element defines a static relationship between two interacting roleTypes. The syntax of relationshipType is following:

```
<relationshipType name="NCName">
   <roleType typeRef="QName" behavior="list of NCName"? />
   <roleType typeRef="QName" behavior="list of NCName"? />
</relationshipType>
```

The name attribute of the relatioshipType is simply a holder for a name. The two nested roleType elements provide references to the roleTypes engaged in the relationship, optionally together with subsets of behaviours they supports. If no behaviours are listed, then it is assumed that all behaviours declared for a roleType are applicable.

13.3.5. participantType

The <participantType> element defines an observable behavior a collaborating participant must provide. The declaration simply enumerates one or more roleTypes assigned to this particular participantType. Such definition is similar to a class declaration with a list of all interfaces this class implements. The syntax of participantType is following:

```
<participantType name="NCName">
   <roleType typeRef="QName" />+
</participantType>
```

where name attribut holds a participantType name, and typeRef attribute of each nested roleType element refer to roleType declarations enumerating supported behaviors.

It is important to understand that it is a participant's duty to maintain the state information associated with each roleType it exposes. If two or more of the roleTypes have a variable with the same name, then they refer to the same piece of information.

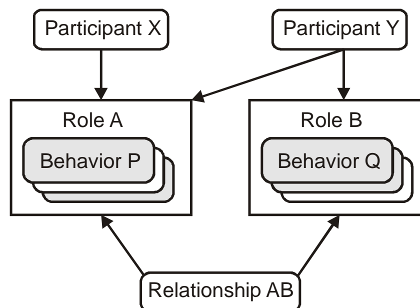


Figure 13.5. Dependences between participant, role, behaviour and relationship.

13.3.6. channelType

A <channelType> element defines an abstract model of a channel – a point of collaboration between participants. It provides details on where and how participants exchange information. A channelType can also restrict the use of its instances, which may be captured within channel variables and passed among participants in information exchanges. These restrictions can be statically checked to ensure that these instances are used in choreography correctly. Thus, in a complex choreography the instances, whether of the same or different channel type, should have own identity. The channelType definition provides solution to that issue. The syntax of their channelType is following:

```
<channelType  name="NCName"
              usage="once"|"distinct"|"shared"?
              action="request-respond"|"request"|"respond"? >

   <passing  channel="QName"
             action="request-respond"|"request"|"respond"?
             new="true"|"false"? />*

   <roleType typeRef="QName"  behavior="NCName"? />

   <reference>
      <token name="QName"/>
   </reference>

   <identity usage="primary"|"alternate"|"derived"|"association">
      <token name="QName"/>+
   <identity>*
</channelType>
```

where name attribute of the channelType holds its name, usage – identifies the permitted usage pattern for the channel instances of this channelType ("once" means that a channel instance can be used for one interaction, or can be passed to another roleType; "distinct" – that a channel instance can be used multiple times by a participantType within multiple interactions and it is a default usage mode; "shared" – that a channel instance can be used multiple times by multiple participantTypes within multiple interactions), action – defines whether the channelType supports "request/response" or just "request", which is default, or just "response" messages.

The elements nested in the channelType serve for the following purposes:

passing – defines the valid type of channel instances passed from one participant to another when using an information exchange on a channel instance of this channelType. The attribute channel refers the definition of a valid channelType of the instance being passed. The optional attribute action defines whether the channel instance passing can occur during a request, which is default, or response or both. The optional attribute new, when set to "true", enforces a passed channel instance to be always unique. If the element passing is missing, then this channelType may be used for exchanging information, but must not be used for passing channel instances of any channelType.

roleType – identifies the destination roleType associated with this channelType. If the action within channelType holds "request" value, then the roleType identified is effectively the provider of the service. In general, the roleType element nested in channelType is used for statically determining where and how to send or receive

information to or from the participant. The meaning of `roleType`'s attributes is following: `typeRef` – holds a name of the destination `roleType` defined in the choreography package; `behavior` – identifies a specific observable behaviour of the destination `roleType` that this channel type is associated with. The `behavior` attribute is optional. If it is missing, then any one of the behaviour types belonging to the `roleType` identified may be a target of an information exchange.

`reference` – identifies the type of the service endpoint reference. The content of this element allows dynamic determination of where and how to send or receive information to or from a participant. The type of a participant reference is distinguished by a token, as specified by the `name` attribute of the nested `token` element.

`identity` – associates a unique identity for a particular `channelType` instance. Each identity is defined by the `name` attribute of the nested `token`. The mandatory `usage` attribute within `identity` defines the purpose of this identity in the context of the `channelType`. The value of this attribute can be:

- `"primary"` – which means that the identity is created by the initial message on an instance of this channelType (only one `"primary"` identity field can be defined in a `channelType`, and all messages exchanged in the instance of the `chanelType` must have the same field);
- `"alternate"` – which means that an alternative identity for a `channelType` instance can be established (such identity can be initialized based on any message within the channel instance's conversation. However, to ensure that it is associated with the appropriate channel instance, this identity must occur in a message that has already bound this identity to the channel instance as it contained either the channel instance's primary key, or another previously initialized alternate identity. Once the alternate identity has been bound to the channel instance, subsequent messages that only contain this alternate identity will be correlated to the right channel instance);
- `"derived"` – which means that the identity will be derived from a message sent on the current channel instance (conversation). Declaration of derived identity type allows establishing a correlation between different channel instances (of the same or different `channelType`) ensuring that they are bound to the same choreography session. Thus, if a message is exchanged on an instance of the `channelType` containing the appropriate information for the derived identity, then the identity will be associated with the choreography session. Then if another channel instance (of the same or different `channelType`) will reference this identity value in its `"primary"` identity field, the referencing channel instance will become correlated with the current channel instance of the same choreography session.
- `"association"` – which means that this channel instance is correlated to a previous channel instance identity, and therefore is associated with the same choreography instance as the previous channel instance. This kind of identity type can be considered equivalent to a backward reference. With an association identity, the initial `channelType` only defines its own identity information, and the subsequent `channelType` (that needs to be correlated to the initial `channelType`) provides the association identity element with the information necessary to map onto one of the primary or alternate identity values in the initial `channelType`.

> NOTE
>
> An identity mechanism in the WS-CDL solves a problem of a proper collaboration of choreography participants based on observed information exchange. Assume, for example, that one participant observed an "Order" request followed by a "Buy" request. How this participant can be sure that both messages belong to the same choreography session, and not to different choreography sessions (or instances)? The answer is hidden in the content of information exchanged. Each message could have a field (or fields) playing a role of identifier(s) and each field could be aliased by a `token`. It would be possible now to create a `tokenLocator` for each message type and a `token` pair to extract each identifier. Although each `tokenLocator` could be created based on knowledge of the `token` and message types to be handled, one way to ensure that all of the appropriate locators are created is to associate the identity token with the channel types. In WS-CDL such association indicates that messages being exchanged over a particular channel type must provide a locator for the token associated with that channel type.

> NOTE
>
> The `channelType` instances exchanged may be used in subsequent interaction activities. This allows the modelling of both static and dynamic message destinations when collaborating within choreography. For example, a "Buyer" could specify `channelType` instance information to be used for sending delivery information. The "Buyer" could then send this `channelType` instance information to a "Seller" who then forwards it to a "Shipper". The "Shipper" could then send delivery information directly to the "Buyer" using the `channelType` instance information originally supplied by the "Buyer". This, of course, requires declaration of proper `relationshipTypes` associating `roleTypes` (and collaborating participants as a consequence).

13.3.7. Choreography

A `<choreography>` element groups a set of interactions into a meaningful business transaction. It consists of variables, activities, exception handler (that specifies what additional actions should occur when a choreography behaves in an abnormal way), finalizer (that specifies additional actions that should occur to modify the effect of an earlier successfully completed choreography, for example, to confirm or undo the effect), and other choreographies.

Choreographies represent modules that can be recursively combined to form new choreographies, using the `perform` activity. This enables more comprehensive business protocols to be assembled from simplier protocol units.

Choreographies may be defined locally or globally, and may be coordinated. The choreography must contain at least one `relationship` type, enumerating the observable behaviour the choreography requires its participants to exhibit. It must contain at least a single activity with some actions performing work.

The choreography life-line expresses the progression of a collaboration. Initially, the collaboration is established between participants, then work is performed within it and finally it completes either normally or abnormally. During its life-time choreography enters states: Initiated State, Enabled State, Unsuccessfully Completed State, Successfully Completed State, Closed State.

The syntax of `choreography` is following:

```
<choreography  name="NCName"
      complete="xsd:boolean XPath-expression"?
      isolation="true"|"false"?
      root="true"|"false"?
      coordination="true"|"false"? >

   <relationship  type="QName" />+

   variableDefinitions?
   Choreography-Notation*
   Activity-Notation

   <exceptionBlock  name="NCName">
      WorkUnit-Notation+
   </exceptionBlock>?

   <finalizerBlock  name="NCName">
      Activity-Notation
   </finalizerBlock>*
</choreography>
```

where `name` defines a distinct name of the choreography, `complete` – is an optional attribute that holds XPath expression used to determine when this choreography can be prematurely completed in a successful state (it is a Boolean conditional), `isolation` – is an optional Boolean property which specifies whether variables defined in the enclosing choreography and changed within enclosed choreography will be available to its sibling choreographies (setting it to true means that the enclosing and sibling choreographies will block when accessing those variables, until the isolated choreography has completed), `coordination` – is a boolean property which if set to true indicates that all participants involved in the choreography will be coordinated so that they all complete at the same time and with the same status, `root` – is a boolean property which identifies whether this is the root choreography (only one top level choreography can be declared as root, which identifies it as the entry point into the set of choreographies included in the CDL package).

The elements nested in the `choreography` serve for the purposes described below:

`relationship` – enumerates the relationships the choreography may participate in (if not defined, then the choreography will be implicitly associated with all relationships); `variableDefinitions` – enumerates the variables defined in this choreography; `Choreography-Notation` – defines the locally defined choreographies that may be performed only within this choreography; `Activity-Notation` – is a place for definitions of actions that perform work; `exeptionBlock` and `finalizerBlock` – are both optional elements used for, respectively, specification of one or more exception workunits, and specification of finalizer activity (a choreography may have more than one `finalizerBlock`, each with one finalizer activity).

13.3.8. Variables

Variables are required for coordination of roles' actions taken within a choreography. They are declared within the scope of `variableDefinitions` as `variable` elements.

Variables allow knowledge sharing about actual states roles entered while executing a choreography. They contain information about commonly observable objects. Values of variables are available to all roles by initializing them prior to the start of a choreography. The variables can be categorized with respect to their use as follows:

– information exchange capturing variables (contain information used when sending or receiving messages),
– state capturing variables (contain information about observable changes of a role as a result of information being exchanged),
– channel capturing variables (contain information such as endpoint URL and quality of service (QoS) for a given channel)
– exception capturing variables (used in case of an exception).

Each variable represents a value declared in a choreography for a specific role. The values of variables can be of `informationType` or `channelType` (mutually exclusive), available to `roleTypes` within a choreography, shared between different `roleTypes` that are part of the same `participantType` (under condition that the variables have the same name). The syntax of `variableDefinitions` is following:

```
<variableDefinitions>
   <variable   name="NCName"
       informationType="QName"?|channelType="QName"?
       mutable="true|false"?
       free="true|false"?
       silent="true|false"?
       roleTypes="list of QName"? />+
</variableDefinitions>
```

where the attribute `name` is used to specify a distinct name for each variable; `informationType` and `channelType` – are mutually exclusive attributes used, respectively, in either information exchange capturing variables or state capturing variables declaration, and in channel capturing variables declaration; `mutable` – is an attribute that specifies whether variable information can change once initialized (if set to "false", the variable information cannot change once initialized); `silent` –specifies if there should be an activity used for creating or changing this variable in the choreography (if set to "true", the manipulation of the value associated with this variable is non-observable); `free` – when set to "true" specifies that a variable is shared by enclosing and current choreographies. Any `perform` activity must bind a free variable defined in a performed choreography with a variable defined in a performing choreography. The variable must be bound to an equivalent (i.e. same type) variable within an enclosing choreography (and cannot be used in its own right without being bound); `roleTypes` – attribute that specifies an XML Schema list of one or more `roleTypes` of a participant at which the variable information will reside (if no list is provided, then the variable will be declared at all roles associated with the choreography).

13.3.9. Activity notation

Activity notations describe actions performed within a choreography in the form of:

`OrderingStructures` – they combine activities with other ordering structures in a nested way to express the ordering rules of actions performed within a choreography using `sequence`, `parallel`, and `choice` elements.

`WorkUnit-Notation` – is used to guard and/or provide a means of repetition of activities enclosed within it (as, for examle, condition and repetition for activity, optionally blocking on data availability)

`BasicActivities` – are used to describe the basic actions performed within a choreography (`interaction`, `perform`, `assign`, `silentAction`, `noAction`,

finalize). The actual work in choreography specification is described in the <activity> elements.

13.3.10. Ordering structures

The ordering structures sequence, parallel and choice are constructs that are used to organize set of activities in blocks and have the following syntax:

```
<sequence>
    Activity-Notation+
</sequence>

<parallel>
    Activity-Notation+
</parallel>

<choice>
    Activity-Notation+
</choice>
```

sequence activity specifies that the set of activities should be performed one by one, in the order they are defined, and no subsequent activity can be performed before its proceeding activity has completed. This activity completes when the final activity in its list has completed.

parallel activity specifies that the activities it contains should be performed concurrently. This activity completes when all of the contained activities have completed.

choice activity specifies two or more mutually exclusive paths of choreography. The proper path is selected based on the observable or non-observable features. These features may be defined as expressions that operate on the state managed by the choreography (i.e. variables), and may relate to exchange events (e.g. messages). A typical use case of the choice construct is to represent alternate paths for a normal or fault response generation.

13.3.11. WorkUnit-Notation

workunit activity represents guarded activity – an activity that includes other choreography structuring constructs and is executed only when a guard condition evaluates to true. This activity provides means of repetition and conditionality. Thus, group of activities can be executed only if some declared pre or post conditions are met.

workunit can be used to model interaction branching by introducing non-observable conditionals for observable exchanges. It serves as a construct that assures consistency of the collaborations commonly performed, and allows recovery from errors.

The guard condition can be configured to wait for all its data to become available. When the variable is/becomes available and the guard condition evaluates to true, the enclosed activities are enabled. The guard condition may be set on variables that reside on different roles.

The syntax of workunit is following:

```
<workunit name="NCName"
      guard="xsd:boolean XPath-expression"?
      repeat="xsd:boolean XPath-expression"?
      block="true|false"? >

      Activity-Notation
</workunit>
```

where name – specifies a name for the workunit, guard – specifies the guard (Boolean condition), repeat – specifies the repetition (Boolean condition), block – specifies whether the workunit has to block waiting for referenced variables within the guard condition to become available (if they are not already) and the guard condition to evaluate to "true".

13.3.12. Interaction activity

interaction is a basic activity. It represents an exchange between roles within a relationship. It encapsulates both sending of messages and subsequent receiving of these messages as a single concept. Any interaction can have a time limit, as well as alignment requirements that ensure synchronization. A specific interaction can be designed to be a choreography initiator. Such interaction activity starts choreography with a message sent from one role to another role through a common channel (and ends with a normal response message or a fault message in a request-response exchange pattern).

A declaration of interaction includes: the channel capturing variable (indicating where and how the message is to be sent to and received into the accepting roleType); the operation (that specifies what a recipient of the message should do with the message when it is received); the participating roles (refered as 'from' and 'to') and the information exchange capturing variables at these roles; informationType or channelType that is being exchanged (depending on what is a subject of exchange); the potential state capturing variables.

The syntax of interaction is following:

```
<interaction  name="NCName"
              channelVariable="QName"
              operation="NCName"
              align="true"|"false"?
              initiate="true"|"false"? >
   <participate  relationshipType="QName"
                 fromRoleTypeRef="QName"
                 toRoleTypeRef="QName" />
   <exchange  name="NCName"
              faultName="QName"?
              informationType="QName"?|channelType="QName"?
              action="request"|"respond" >
     <send      variable="XPath-expression"?
                recordReference="list of NCName"?
                causeException="QName"? />
     <receive   variable="XPath-expression"?
                recordReference="list of NCName"?
                causeException="QName"? />
   </exchange>*
   <timeout  time-to-complete="XPath-expression"
             fromRoleTypeRecordRef="list of NCName"?
             toRoleTypeRecordRef="list of NCName"? />?
   <record  name="NCName"
            when="before"|"after"|"timeout"
            causeException="QName"? >
   <source variable="XPath-expression"? | expression="XPath-expression"? />
   <target  variable="XPath-expression" />
   </record>*
</interaction>
```

where name is a short name used to describe the interaction, channel – refers to the variable that represents the channelType instance on which communication is performed; operation – specifies a name of the operation that is associated with this interaction (operation belongs to the interface, as identified by the roleType and behaviour elements specified in the channel variable used in this interaction); align and initiate – are

Boolean attributes, indicating, respectively, whether the result of this interaction should be aligned with the communicating endpoints (i.e. they both agree that they have the same resulting state before the interaction completes), and whether this interaction is responsible for initiating the choreography.

The elements nested in the `interaction` serve for the purposes as described below:

`participate` – includes `relationshipType` attribute that specifies the relationshipType this interaction participates in and two mode attributes: `fromRoleTypeRef` and `toRoleTypeRef`, identifying the requesting and the accepting roleTypes (accepting `roleType` must be the same as the `roleType` identified by the `roleType` element specified in the `channel` variable)

`exchange` – optional element that allows information to be exchanged during an interaction. This element includes several optional attributes (`name` – used to name this exchange; `faultName` – used to identify this exchange element as a fault exchange with the specified name; `informationType` and `channelType` – mutually exclusive attributes identifying, respectively, the informationType and the channelType of the information that is exchanged between the two roleTypes) and `action` attribute, that specifies the direction of the information exchange. It can include also two nested elements: `send` – that shows that information is sent from a roleType, and `receive` – that shows that information is received at a roleType in the interaction.

`timeout` – defines a timeframe within which an interaction must complete after it was initiated, or the deadline before which an interaction must complete (`time-to-complete` provides time limits; `fromRoleTypeRecordRef` / `toRoleTypeRecordRef` provides a list of records in the same interaction that will be performed at the 'from'/'to' roleType when a timeout occurs).

`record` – is used to create a value of one or more variables using another variable or an expression (nested `source` element must define either a variable attribute or an expression attribute) or change and then make available variable or expression within one roleType (`name` – specifies a distinct record's name, `when` – defines when the recording happens, `causeException` – identifies an exception that may be caused at the corresponding roleType). Nested `source` and `target` elements specify recordings of information happening at the send and receive ends of the interaction.

13.3.13. Perform activity

`perform` is a basic activity allowing composition of choreographies by their combination. It is used to indicate that the activities within a referenced choreography should occur at certain point and that referenced choreographies are either directly defined within the performing choreography, or at the top level. It provides also optional capability to bind variables within the performing choreography with variables in the performed choreography. When this feature is used, it means that perform has no direct storage of its own, but instead maps onto the storage of a variable contained within an enclosing choreography.

The syntax of `perform` is following:

```
<perform  choreographyName="QName"
          choreographyInstanceId="XPath-expression"?
          block="true|false"? >
 <bind  name="NCName">
   <this variable="XPath-expression" roleType="QName"/>
   <free variable="XPath-expression" roleType="QName"/>
 </bind>*
```

```
 Choreography-Notation?
</perform>
```

where `choreographyName` – defines a reference to the choreography that is being performed, `choreographyInstanceId` – identifies a specific instance of the choreography to be performed in this performance in cases where multiple instances of this choreography will be performed and later finalized, `block` – determines whether the performing choreography should wait for the performed choreography to complete, before preceding to the next activity.

The `bind` element, nested in the `perform`, enables information sharing between performing and performed choreography. This element can be named using `name` attribute. It includes `this` element (that defines the `variable` within the performing choreography, that will be bound with the `variable` identified within the free element in the performed choreography, and aliases `roleType` from the performing choreography to the performed choreography) and `free` element (that defines `variable`, within the performed choreography, that will be bound, and `roleType` that is reverted `roleType` from `this` element within this choreography).

`Choreography-Notation` – defines a locally defined choreography that is performed only by this `perform` activity.

13.3.14. Assign activity

`assign` is a basic activity which provides a mechanism for assigning values to one or more variables within one `roleType` using the value of another variable or expression (what may cause an exception at a `roleType`). Assignments perform at the declared role type. If the source information is unavailable, the result of an assignment is undefined.

The syntax of `assign` is following:

```
<assign  roleType="QName">
 <copy  name="NCName" causeException="QName"? >
   <source variable="XPath-expression"?| expression="XPath-expression"? />
   <target  variable="XPath-expression" />
 </copy>+
</assign>
```

where `name` is the name of this assignment. The `copy` element nested in `assign` (`named` and with indication that it may `causeException`) includes `source` – element that defines the source variable through its reference or XPath expression to derive the value that will be copied into the target variable (the use of `variable` or `expression` should be mutually exclusive), and `target` – element that defines the target variable into which the copied value will be assigned.

13.3.15. SilentAction activity

This basic activity provides an explicit designator used for specifying the point where participant specific actions with non-observable operational details are performed. These actions may modify the state information associated with a particular participant and role. However, it will not be entitled to perform interactions, as this would interfere with the externally observable behaviour of the role.

The syntax of `silentAction` is following:

```
<silentAction roleType="QName"? />
```

where `roleType` is used to specify the participant at which the `silentAction` will be performed.

13.3.16. NoAction activity

This basic activity defines that the participants involved within choreography should not perform any internal tasks that would modify the observable behaviour exhibited by those participants. Thus, within this activity the participant would not be able to perform any interactions, or modify state information that is defined within choreography.

The syntax of `noAction` is following:

```
<noAction roleType="QName"? />
```

where `roleType` is used to specify a participant at which the `noAction` will be performed.

13.3.17. Finalize activity

This basic activity is used to initiate the finalization handling (to enable a particular `finalizerBlock`) for a choreography that has previously been performed (using the `perform` activity) and thus concludes choreography.

The syntax of `finalize` is following:

```
<finalize  name="NCName"? choreographyName="NCName"
            choreographyInstanceId="XPath-expression"?
            finalizerName="NCName"? />
```

where `name` – specifies a distinct name for each finalize element, `choreographyName` – identifies the choreography referenced by the `choreographyName` attribute of the perform construct, `choreographyInstanceId` – identifies the performed choreography instance to be finalized, using the value defined by the `choreographyInstanceId` attribute of the perform construct, `finalizerName` – indicates which `finalizerBlock` is to be enabled in the performed instance.

13.3.18. Exception block

WS-CDL provides a mechanism to enable choreography to terminate in case of occurrence of an unusual situation caused by: interaction failures, protocol based exchange failures, security failures, timeout errors, validation errors, application failures etc.

Some choreography activities have a property, called cause exception, to indicate that when the activity is performed, then an exception of the type associated with the activity should be raised. The exceptions are handled within `exceptionBlocks` defined as workUnits (with a guard condition defined using `hasExceptionOccurred` function). One or more exception workunits may be defined for each exception that needs to be handled. The exception handlers' properties define names and descriptions of the exception handling activities. They also allow selecting the information type associated with the exception.

13.3.19. Predefined functions

Within expressions defined in choreography, it is possible to make use of a set of pre-defined functions. Functions are simply invoked by expressing the function name (including the namespace) and the set of comma separated parameters within the following brackets. The list of predefined functions contains:

`any getChannelIdentity(string varName)` – is used to get the identity information associated with a channel instance.

`any getChannelReference(string varName)` – is used to get the endpoint reference information associated with a channel instance.

`time getCurrentTime(QName roleName)` – is used to retrieve the current time at the role identified by the `roleName` parameter.

`date getCurrentDate(QName roleName)` – is used to retrieve the current date at the role identified by the `roleName` parameter.

`dateTime getCurrentDateTime(QName roleName)` – is used to retrieve the current date and time. This function will be performed locally at the role identified by the `roleName` parameter.

`any getVariable(string varName, string part, string documentPath, QName roleName?)` – is used to obtain a value of a named variable at the specific `roleType`. If the variable is a WSDL1.1 message, then it will also be necessary to identify the part of the message that should be retrieved.

`boolean isVariableAvailable(string varName, QName roleName)` – is used to check if a variable is already available at a specific roleType or to wait for a variable to become available at a specific `roleType`, based on the block attribute being `"false"` or `"true"` respectively.

`boolean variablesAligned(string varName, string withVarName, QName relationshipName)` – is used to check or wait for an appropriate alignment interaction to happen between the two `roleTypes`, based on the block attribute being `"false"` or `"true"` respectively. Alignment means that there has been an agreement that the two variables have the same value.

`boolean globalizedTrigger(string expression, string roleName, string expression2, string roleName2, ...)` – is used in case of combining constraints prescribed for each `roleType` but without requiring that all these constraints have to be fulfilled for progress to be made.

`boolean hasDeadlinePassed(dateTime deadlineTime, QName roleName)` – is used to determine whether a particular deadline time has passed.

`boolean hasDurationPassed(duration elapsedTime, QName roleName)` – is used to determine whether a particular time interval has passed.

`boolean hasExceptionOccurred(QName exceptionType)` – is used to determine whether an exception of the type identified by the parameter exceptionType has occurred.

`boolean hasChoreographyCompleted(string choreoName, string choreoInstanceId?)` – is used to determine whether the performed choreography associated with the parameter `choreoName` and optional `choreoInstanceId` has a status of completed (whether successfully or not).

`string getChoreographyStatus(string choreoName, string choreoInstanceId?)` – is used to get the current status associated with the identified choreography by `choreoName` and optional `choreoInstanceId`.

13.3.20. WS-CDL Example

The example of a WS-CDL document provided below comes from the pi4soa project (available at http://pi4soa.sourceforge.net). In this document there are three roles defined: Buyer, Seller and CreditAgency. Buyer sends the BuyRequest to the Seller. Seller asks CreditAgency to check the credit with CreditCheck request. If credit is OK, then the CreditAgency sends back to the seller CreditCheckOk message. If not, the message sent is CreditCheckFailed. Seller, depending on the response from CreditAgency, confirms the Buyer

order sending BuyConfirmed message, or rejects it sending BuyFailed message. The schema of this choreography was shown in Figure 13.1. The following code is the choreography description expressed in WS-CDL language.

```
<?xml version="1.0" encoding="Cp1252"?>
<org.pi4soa.cdl:Package xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:org.pi4soa.cdl="http:///org/pi4soa/cdl.ecore" name="BuyerSellerCDL"
author="Steve Ross-Talbot" version="1.4"
targetNamespace="http://www.pi4tech.com/cdl/BuyerSeller">
  <typeDefinitions>
    <nameSpaces prefix="bs"
uRI="http://www.pi4tech.com/cdl/BuyerSellerExample-1"/>
    <nameSpaces prefix="xsd" uRI="http://www.w3.org/2001/XMLSchema"/>
    <informationTypes name="BooleanType" typeName="xsd:boolean"
elementName=""/>
    <informationTypes name="CreditAcceptType" typeName="bs:CreditAccept"
elementName=""/>
    <informationTypes name="CreditCheckType"
typeName="bs:CreditCheckRequest" elementName=""/>
    <informationTypes name="CreditRejectType" typeName="bs:CreditReject"
elementName=""/>
    <informationTypes name="DeliveryDetailsType"
typeName="bs:DeliveryDetails" elementName=""/>
    <informationTypes name="QuoteAcceptType" typeName="bs:QuoteAccept"
elementName=""/>
    <informationTypes name="QuoteType" typeName="bs:Quote" elementName=""/>
    <informationTypes name="QuoteUpdateType" typeName="bs:QuoteUpdate"
elementName=""/>
    <informationTypes name="RequestDeliveryType"
typeName="bs:RequestForDelivery" elementName=""/>
    <informationTypes name="RequestForQuoteType"
typeName="bs:RequestForQuote" elementName=""/>
    <informationTypes name="StringType" typeName="xsd:string"
elementName=""/>
    <tokens name="BuyerRef"
informationType="//@typeDefinitions/@informationTypes.10"/>
    <tokens name="CreditCheckRef"
informationType="//@typeDefinitions/@informationTypes.10"/>
    <tokens name="SellerRef"
informationType="//@typeDefinitions/@informationTypes.10"/>
    <tokens name="ShipperRef"
informationType="//@typeDefinitions/@informationTypes.10"/>
    <roleTypes name="BuyerRoleType">
      <behaviors name="BuyerBehavior" interface=""/>
    </roleTypes>
    <roleTypes name="CreditCheckerRoleType">
      <behaviors name="CreditCheckerBehavior" interface=""/>
    </roleTypes>
    <roleTypes name="SellerRoleType">
      <behaviors name="SellerBehavior" interface=""/>
    </roleTypes>
    <roleTypes name="ShipperRoleType">
      <behaviors name="ShipperBehavior" interface=""/>
    </roleTypes>
    <relationshipTypes name="BuyerSeller"
firstRoleType="//@typeDefinitions/@roleTypes.0"
secondRoleType="//@typeDefinitions/@roleTypes.2"/>
```

```
    <relationshipTypes name="SellerCreditCheck"
firstRoleType="//@typeDefinitions/@roleTypes.2"
secondRoleType="//@typeDefinitions/@roleTypes.1"/>
    <relationshipTypes name="SellerShipper"
firstRoleType="//@typeDefinitions/@roleTypes.2"
secondRoleType="//@typeDefinitions/@roleTypes.3"/>
    <relationshipTypes name="ShipperBuyer"
firstRoleType="//@typeDefinitions/@roleTypes.3"
secondRoleType="//@typeDefinitions/@roleTypes.0"/>
    <participantTypes name="Buyer"
roleTypes="//@typeDefinitions/@roleTypes.0"/>
    <participantTypes name="CreditChecker"
roleTypes="//@typeDefinitions/@roleTypes.1"/>
    <participantTypes name="Seller"
roleTypes="//@typeDefinitions/@roleTypes.2"/>
    <participantTypes name="Shipper"
roleTypes="//@typeDefinitions/@roleTypes.3"/>
    <channelTypes name="2BuyerChannelType" action="Request"
referenceToken="//@typeDefinitions/@tokens.0"
roleType="//@typeDefinitions/@roleTypes.0"/>
    <channelTypes name="Buyer2SellerChannelType"
referenceToken="//@typeDefinitions/@tokens.2"
roleType="//@typeDefinitions/@roleTypes.2">
      <passingChannelDetails channel="//@typeDefinitions/@channelTypes.0"
new="true"/>
    </channelTypes>
    <channelTypes name="Seller2CreditCheckChannelType"
referenceToken="//@typeDefinitions/@tokens.1"
roleType="//@typeDefinitions/@roleTypes.1"/>
    <channelTypes name="Seller2ShipperChannelType"
referenceToken="//@typeDefinitions/@tokens.3"
roleType="//@typeDefinitions/@roleTypes.3">
      <passingChannelDetails channel="//@typeDefinitions/@channelTypes.0"/>
    </channelTypes>
  </typeDefinitions>
  <choreographies name="Main" completionCondition="" root="true"
relationships="//@typeDefinitions/@relationshipTypes.0
//@typeDefinitions/@relationshipTypes.1
//@typeDefinitions/@relationshipTypes.2
//@typeDefinitions/@relationshipTypes.3">
    <variableDefinitions name="Buyer2SellerC"
type="//@typeDefinitions/@channelTypes.1"
roleTypes="//@typeDefinitions/@roleTypes.0
//@typeDefinitions/@roleTypes.2"/>
    <variableDefinitions name="DeliveryDetailsC"
type="//@typeDefinitions/@channelTypes.0"
roleTypes="//@typeDefinitions/@roleTypes.0 //@typeDefinitions/@roleTypes.2
//@typeDefinitions/@roleTypes.3"/>
    <variableDefinitions name="Seller2CreditChkC"
type="//@typeDefinitions/@channelTypes.2"/>
    <variableDefinitions name="Seller2ShipperC"
type="//@typeDefinitions/@channelTypes.3"
roleTypes="//@typeDefinitions/@roleTypes.2
//@typeDefinitions/@roleTypes.3"/>
    <variableDefinitions name="barteringDone"
type="//@typeDefinitions/@informationTypes.0"
roleTypes="//@typeDefinitions/@roleTypes.0
//@typeDefinitions/@roleTypes.2"/>
    <activities xsi:type="org.pi4soa.cdl:Interaction" description="Buyer
requests a Quote - this is the initiator" name="BuyerRequestsQuote"
operation="requestForQuote"
```

```
channelVariable="//@choreographies.0/@variableDefinitions.0"
initiate="true" relationship="//@typeDefinitions/@relationshipTypes.0">
      <exchangeDetails name="request"
type="//@typeDefinitions/@informationTypes.9" sendCauseException=""
receiveCauseException=""/>
      <exchangeDetails name="response"
type="//@typeDefinitions/@informationTypes.6" action="Respond"
sendCauseException="" receiveCauseException=""/>
    </activities>
    <activities xsi:type="org.pi4soa.cdl:While" description="Repeat until
bartering has been completed" name="BarteringLoop" expression=""
reEvaluateCondition="barteringDone = false">
      <activities xsi:type="org.pi4soa.cdl:Choice">
        <activities xsi:type="org.pi4soa.cdl:SilentAction"
roleType="//@typeDefinitions/@roleTypes.0" name="NoBartering"/>
        <activities xsi:type="org.pi4soa.cdl:Sequence">
          <activities xsi:type="org.pi4soa.cdl:Interaction"
description="Buyer accepts the quote and engages in the act of buying"
name="AcceptQuote" operation="quoteAccept"
channelVariable="//@choreographies.0/@variableDefinitions.0"
relationship="//@typeDefinitions/@relationshipTypes.0">
            <exchangeDetails name="AcceptQuote"
type="//@typeDefinitions/@informationTypes.5" sendCauseException=""
receiveCauseException=""/>
          </activities>
          <activities xsi:type="org.pi4soa.cdl:Interaction"
description="Buyer send channel to seller to enable callback behavior"
name="SendChannel" operation="sendChannel"
channelVariable="//@choreographies.0/@variableDefinitions.0"
relationship="//@typeDefinitions/@relationshipTypes.0">
            <exchangeDetails name="sendChannel"
type="//@typeDefinitions/@channelTypes.0"
sendVariable="//@choreographies.0/@variableDefinitions.1"
receiveVariable="//@choreographies.0/@variableDefinitions.1"
sendCauseException="" receiveCauseException=""/>
          </activities>
          <activities xsi:type="org.pi4soa.cdl:Assign"
roleType="//@typeDefinitions/@roleTypes.0">
            <copyDetails name="copy" sourceExpression="true"
targetVariable="//@choreographies.0/@variableDefinitions.4"
causeException=""/>
          </activities>
        </activities>
        <activities xsi:type="org.pi4soa.cdl:Sequence">
          <activities xsi:type="org.pi4soa.cdl:Interaction"
description="Buyer updates the Quote - in effect requesting a new price"
name="RequestNewPrice" operation="quoteUpdate"
channelVariable="//@choreographies.0/@variableDefinitions.0"
relationship="//@typeDefinitions/@relationshipTypes.0">
            <exchangeDetails name="updateQuote"
type="//@typeDefinitions/@informationTypes.7" sendCauseException=""
receiveCauseException=""/>
            <exchangeDetails name="acceptUpdatedQuote"
type="//@typeDefinitions/@informationTypes.5" action="Respond"
sendCauseException="" receiveCauseException=""/>
          </activities>
        </activities>
      </activities>
    </activities>
    <activities xsi:type="org.pi4soa.cdl:Interaction" description="Seller
check credit with CreditChecker" name="CreditCheck" operation="creditCheck"
```

```
channelVariable="//@choreographies.0/@variableDefinitions.2"
relationship="//@typeDefinitions/@relationshipTypes.1">
      <exchangeDetails name="checkCredit"
type="//@typeDefinitions/@informationTypes.2" sendCauseException=""
receiveCauseException=""/>
    </activities>
    <activities xsi:type="org.pi4soa.cdl:Choice">
      <activities xsi:type="org.pi4soa.cdl:Interaction" description="Credit
Checker fails credit check" name="CheckFails" operation="creditCheck"
channelVariable="//@choreographies.0/@variableDefinitions.2"
relationship="//@typeDefinitions/@relationshipTypes.1">
        <exchangeDetails name="creditCheckFails"
type="//@typeDefinitions/@informationTypes.3" action="Respond"
sendCauseException="" receiveCauseException=""
faultName="creditCheckFails"/>
      </activities>
      <activities xsi:type="org.pi4soa.cdl:Sequence">
        <activities xsi:type="org.pi4soa.cdl:Interaction"
description="Credit Checker passes credit" name="CreditOk"
operation="creditCheck"
channelVariable="//@choreographies.0/@variableDefinitions.2"
relationship="//@typeDefinitions/@relationshipTypes.1">
          <exchangeDetails name="creditCheckPasses"
type="//@typeDefinitions/@informationTypes.1" action="Respond"
sendCauseException="" receiveCauseException=""/>
        </activities>
        <activities xsi:type="org.pi4soa.cdl:Interaction"
description="Seller requests delivery details - passing channel for buyer
and shipper to interact" name="ReqDelivery" operation="requestShipping"
channelVariable="//@choreographies.0/@variableDefinitions.3"
relationship="//@typeDefinitions/@relationshipTypes.2">
          <exchangeDetails name="sellerRequestsDelivery"
type="//@typeDefinitions/@informationTypes.8" sendCauseException=""
receiveCauseException=""/>
          <exchangeDetails name="sellerReturnsDelivery"
type="//@typeDefinitions/@informationTypes.4" action="Respond"
sendCauseException="" receiveCauseException=""/>
        </activities>
        <activities xsi:type="org.pi4soa.cdl:Interaction"
description="Shipper forward channel to shipper" name="SendChannel"
operation="sendChannel"
channelVariable="//@choreographies.0/@variableDefinitions.3"
relationship="//@typeDefinitions/@relationshipTypes.2">
          <exchangeDetails name="forwardChannel"
type="//@typeDefinitions/@channelTypes.0"
sendVariable="//@choreographies.0/@variableDefinitions.1"
receiveVariable="//@choreographies.0/@variableDefinitions.1"
sendCauseException="" receiveCauseException=""/>
        </activities>
        <activities xsi:type="org.pi4soa.cdl:Interaction"
description="Shipper sends delivery details to buyer"
name="DeliveryDetails" operation="deliveryDetails"
channelVariable="//@choreographies.0/@variableDefinitions.1"
relationship="//@typeDefinitions/@relationshipTypes.3">
          <exchangeDetails name="sendDeliveryDetails"
type="//@typeDefinitions/@informationTypes.4" sendCauseException=""
receiveCauseException=""/>
        </activities>
      </activities>
    </activities>
  </choreographies>
```

</org.pi4soa.cdl:Package>

Literature

1. A.B. Bondi: Characteristics of scalability and their impact on performance, Proceedings of the 2nd international workshop on Software and performance, Ottawa, Ontario, Canada, 2000, ISBN 1-58113-195-X, pages 195 – 203
2. A. Deepak, J. Crupi, D. Malks, Core J2EE Patterns: Best Practices and Design Strategies, 2nd Edition, Prentice Hall Ptr, 2003
3. E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1994.
4. I. Jacobson, G. Booch, J. Rumbaugh: The Unified Software Development Process, Addison-Wesley Professional, 1999
5. J. Nilson: Applying Domain-Driven Design and Patterns with Examples in C# and .NET, Addison-Wesley Professional, 2006
6. R.C. Martin, M. Martin: AGILE principles, patterns and practices in C#, Prentice Hall, 2006
7. P. Norvig: Design Patterns in Dynamic Programming (1998-03-17), http://norvig.com/design-patterns.
8. A. Shalloway, J. Trott: Design Patterns Explained: A New Perspective on Object-Oriented Design, Addison-Wesley Professional, 2004
9. Tutorial Java EE 6.0. Available at: http://download.oracle.com/javaee/6/tutorial/doc/bnaay.html
10. E. Newcomer, Understanding Web Services: XML, WSDL, SOAP, and UDDI
11. S. Weerawarana, F. Curbera, F. Leymann, T. Storey. Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More
12. T. Berners-Lee: Notation3 (N3) A Readable RDF syntax, World Wide Web Consortium. Available at: http://www.w3.org/DesignIssues/Notation3.html
13. METEOR-S: Semantic Web Services and Processes. Available at: http://lsdis.cs.uga.edu/projects/meteor-s/
14. OWL 2 Web Ontology Language: Conformance, M. Smith, I. Horrocks, M. Krötzsch, B. Glimm (eds.) W3C Recommendation, 27 October 2009. The latest version available at http://www.w3.org/TR/owl2-conformance/.
15. OWL 2 Web Ontology Language Data Range Extension: Linear Equations, B. Parsia, U. Sattler, W3C Working Group Note 27 October 2009. The latest version is available at: http://www.w3.org/TR/owl2-dr-linear/
16. OWL 2 Web Ontology Language: Direct Semantics, B. Motik, P.F. Patel-Schneider, B.C. Grau (eds.) W3C Recommendation, 27 October 2009. The latest version is available at: http://www.w3.org/TR/owl2-direct-semantics/.
17. OWL 2 Web Ontology Language Manchester Syntax, M. Horridge, P.F. Patel-Schneider, W3C Working Group Note 27 October 2009. The latest version is available at: http://www.w3.org/TR/owl2-manchester-syntax/
18. OWL 2 Web Ontology Language New Features and Rationale, C. Golbreich, E.K. Wallace (eds.) W3C Recommendation 27 October 2009. The latest version (series 2) is available at: http://www.w3.org/TR/owl2-new-features/
19. OWL 2 Web Ontology Language Document Overview, W3C OWL Working Group, eds. W3C Recommendation 27 October 2009. The latest version (series 2) is available at: http://www.w3.org/TR/owl2-overview/ .
20. OWL 2 Web Ontology Language: Primer, P. Hitzler, M. Krötzsch, B. Parsia, P.F. Patel-Schneider, S. Rudolph (eds.) W3C Recommendation, 27 October 2009. The latest version is available at http://www.w3.org/TR/owl2-primer/.
21. OWL 2 Web Ontology Language: Profiles, B. Motik, B.C. Grau, I. Horrocks, Z. Wu, A. Fokoue, C. Lutz (eds.) W3C Recommendation, 27 October 2009, Latest version available at: http://www.w3.org/TR/owl2-profiles/.
22. OWL 2 Web Ontology Language Quick Reference Guide, J. Bao, E.F. Kendall, D.L. McGuinness, P.F. Patel-Schneider (eds.) W3C Recommendation 27 October 2009. The latest version (series 2) is available at: http://www.w3.org/TR/owl2-quick-reference/
23. OWL 2 Web Ontology Language: Mapping to RDF Graphs, P.F. Patel-Schneider, B. Motik, eds. W3C Recommendation, 27 October 2009. The latest version is available at http://www.w3.org/TR/owl2-mapping-to-rdf/.
24. OWL 2 Web Ontology Language: RDF-Based Semantics Michael Schneider, editor. W3C Recommendation, 27 October 2009. The latest version available at: http://www.w3.org/TR/owl2-rdf-based-semantics/.
25. OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax, B. Motik, P.F. Patel-Schneider, B. Parsia (eds.) W3C Recommendation, 27 October 2009. The latest version is available at: http://www.w3.org/TR/owl2-syntax/ .

26. OWL 2 Web Ontology Language: XML Serialization, B. Motik, B. Parsia, P.F. Patel-Schneider (eds.) W3C Recommendation, 27 October 2009, http://www.w3.org/TR/2009/REC-owl2-xml-serialization-20091027/. Latest version available at http://www.w3.org/TR/owl2-xml-serialization/.
27. F. van Harmelen, J. Hendler, I. Horrocks, D.L. McGuinness, P.F. Patel-Schneider, L.A. Stein: OWL Web Ontology Language Reference, Dean M., Schreiber G (eds.) W3C Recommendation, 10 February 2004. The latest version is available at: http://www.w3.org/TR/owl-ref/.
28. Web Ontology Language for Web Services. Available at: http://www.daml.org/services/
29. PRISM: Publishing Requirements for Industry Standard Metadata, Version 1.1, 19 February 2002. The latest version is available at http://www.prismstandard.org/.
30. Resource Description Framework (RDF): Concepts and Abstract Syntax, Klyne G., Carroll J. (eds.), W3C Recommendation, 10 February 2004. The latest version is available at: http://www.w3.org/TR/rdf-concepts/.
31. MIME Media Types, The Internet Assigned Numbers Authority (IANA). Available at: http://www.iana.org/assignments/media-types/ . The registration for application/rdf+xml is archived at http://www.w3.org/2001/sw/RDFCore/mediatype-registration.
32. Resource Description Framework (RDF) Model and Syntax Specification, Lassila O., Swick R. (eds.), World Wide Web Consortium, 22 February 1999. The latest version is available at: http://www.w3.org/TR/REC-rdf-syntax/.
33. RDF Primer. F. Manola, E. Miller (eds.), W3C Recommendation 10 February 2004. Tge latest version is available at: http://www.w3.org/TR/rdf-primer/.
34. RDF Semantics, Hayes P. (ed.), W3C Recommendation, 10 February 2004. The latest version is available at: http://www.w3.org/TR/rdf-mt/.
35. RDF/XML Syntax Specification (Revised), Beckett D. (ed.), W3C Recommendation, 10 February 2004. The latest version is available at: http://www.w3.org/TR/rdf-syntax-grammar/.
36. RDF Test Cases, Grant J., Beckett D. (Editors), W3C Recommendation, 10 February 2004. The latest version is available at: http://www.w3.org/TR/rdf-testcases/.
37. RDF Vocabulary Description Language 1.0: RDF Schema, Brickley D., Guha R.V. (eds.), W3C Recommendation, 10 February 2004. The latest version is available at: http://www.w3.org/TR/rdf-schema/.
38. T. Berners-Lee, R. Fielding, L. Masinter: RFC 2396 - Uniform Resource Identifiers (URI): Generic Syntax, IETF, August 1998. Available at: http://www.ietf.org/rfc/rfc2396.txt.
39. D. Eastlake, A. Panitz: Reserved Top Level DNS Names, June 1999. Available at: http://www.rfc-editor.org/rfc/rfc2606.txt
40. R. Hinden, B. Carpenter, L. Masinter: RFC 2732 - Format for Literal IPv6 Addresses in URLs, IETF, December 1999. Available at: http://www.ietf.org/rfc/rfc2732.txt
41. T. Berners-Lee, R. Fielding, L. Masinter: RFC 3986 - Uniform Resource Identifier (URI): Generic Syntax. IETF, January 2005. Available at: www.ietf.org/rfc/rfc3986.txt
42. M. Duerst, M. Suignard: RFC 3987 - Internationalized Resource Identifiers (IRIs), IETF, January 2005. Available at: http://www.ietf.org/rfc/rfc3987.txt
43. RDF Site Summary (RSS) 1.0, G. Beged-Dov, D. Brickley, R. Dornfest, I. Davis, L. Dodds, J. Eisenzopf, D. Galbraith, R.V. Guha, K. MacLeod, E. Miller, A. Swartz, E. van der Vlist, 2000. Available at: http://purl.org/rss/1.0/spec.
44. The Unicode Standard, Version 3, The Unicode Consortium, Addison-Wesley, 2000. The latest version and additional information ara available at: http://www.unicode.org/unicode/standard/versions/
45. T. Berners-Lee, D. Connolly, R. Swick: Web Architecture: Describing and Exchanging Data, World Wide Web Consortium, 7 June 1999. Available at http://www.w3.org/1999/04/WebData.
46. Web Services Choreography Description Language, W3C Candidate Recommendation 9 November 2005. Available at: http://www.w3.org/TR/ws-cdl-10/
47. Web Services Choreography Description Language: Primer. W3C Working Draft 19 June 2006, Available at: http://www.w3.org/TR/ws-cdl-10-primer/
48. WS Choreography Model Overview, W3C Working Draft 24 March 2004. Available at: http://www.w3.org/TR/ws-chor-model/
49. Web Services Choreography Requirements, W3C Working Draft 11 March 2004. Available at: http://www.w3.org/TR/ws-chor-reqs/
50. Web Services Description Language (WSDL) Version 2.0 Part 2: Adjuncts, R. Chinnici, H. Haas, A. Lewis, J-J. Moreau, D. Orchard, S. Weerawarana (eds.) World Wide Web Consortium, 26 June 2007. The latest version is available at: http://www.w3.org/TR/wsdl20-adjuncts.
51. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language, R. Chinnici, J-J. Moreau, A. Ryman, S. Weerawarana (eds.) World Wide Web Consortium, 26 June 2007. The latest version is available at: http://www.w3.org/TR/wsdl20.
52. Web Services Description Language (WSDL) 1.1, W3C Note, 15 March 2001. Available at: http://www.w3.org/TR/wsdl]

53. WSDL-S: Adding semantics to WSDL - White paper. Available at: http://lsdis.cs.uga.edu/library/download/wsdl-s.pdf
54. Web Service Modelling Language. Available at: http://www.wsmo.org/wsml/
55. Web Service Modelling Ontology. Available at: http://www.wsmo.org/
56. XML Linking Language (XLink) Version 1.0, S. DeRose, E. Maler, D. Orchard (eds.) World Wide Web Consortium, 27 June 2001. The latest version is available at: http://www.w3.org/TR/xlink/.
57. Extensible Markup Language (XML) 1.0, Second Edition, T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler (eds.), World Wide Web Consortium, 6 October 2000. The latest version is available at: http://www.w3.org/TR/REC-xml.
58. XML Base, Marsh J. (ed.), World Wide Web Consortium, 27 June 2001. The latest version is available at http://www.w3.org/TR/xmlbase/.
59. Namespaces in XML, T. Bray, D. Hollander , A. Layman (eds.), World Wide Web Consortium, 14 January 1999. The latest version is available at: http://www.w3.org/TR/REC-xml-names/ .
60. XML Schema Part 2: Datatypes, P. Biron, A. Malhotra(eds.) World Wide Web Consortium. 2 May 2001.The latest version is available at: http://www.w3.org/TR/xmlschema-2/.
61. XML Schema Part 1: Structures Second Edition, H.S. Thompson, D. Beech, M. Maloney, N. Mendelsohn (eds.) W3C Recommendation 28 October 2004. The latest version is available at: http://www.w3.org/TR/xmlschema-1/
62. Exclusive XML Canonicalization Version 1.0, J. Boyer, D.E. Eastlake 3rd, J. Reagle (Authors/Editors), World Wide Web Consortium, 18 July 2002. The latest version is http://www.w3.org/TR/xml-exc-c14n/ .