University of West Bohemia

Faculty of Applied Sciences

Advanced Interactive Visualization Approach for Component-Based Software

Ing. Jaroslav Šnajberk

Doctoral Thesis
in partial fulfillment of the requirements
for the degree of *Doctor of Philosophy*
in specialization *Computer Science and Engineering*

Supervisor: Doc. Ing. Přemysl Brada, MSc. Ph.D.

Pilsen, 2013

Západočeská univerzita v Plzni

Fakulta aplikovaných věd

Advanced Interactive Visualization Approach for Component-Based Software

Ing. Jaroslav Šnajberk

Disertační práce
k získání akademického titulu *doktor*
v oboru *Informatika a výpočetní technika*

Školitel: Doc. Ing. Přemysl Brada, MSc. Ph.D.

V Plzni, 2013

Prohlášení

Předkládám tímto k posouzení a obhajobě disertační práci zpracovanou na závěr doktorského studia na Fakultě aplikovaných věd Západočeské univerzity v Plzni.

Prohlašuji tímto, ze tuto práci jsem vypracoval samostatně, s použitím odborné literatury a dostupných pramenů uvedených v seznamu, jenž je součástí této práce.

V Plzni dne ....   Ing. Jaroslav Šnajberk

Abstract

This thesis deals with the topic of component-based application structure visualization. Use of components can improve organization and clarity of application architectures by encapsulating the implementation details. Software architects can thus work on higher level of abstraction, where they create new applications by simply assembling them from smaller parts. However in recent research only a small amount of focus was given to how these component-based applications could be visualized.

This thesis offers a solution to this deficiency in the form of a new visualization approach called AIVA (Advanced Interactive Visualization Approach), targeted specifically to software components. It works with a sufficient amount of detail about them that can help architects understand all components in an architecture more deeply. At the same time it is focused on how to show these details without increasing the complexity of the final component diagram. Various interactive techniques are used to lower the diagram complexity below a standard level used in other diagramming approaches while providing more information. These techniques are also used to provide all information as soon as possible, resulting in a faster learning process.

As part of this work, the new visualization approach was evaluated using a case study concerning the complexity of the resulting diagrams and by a user study of its performance. The results of these studies showed that AIVA leads to less complex diagrams with a structure that is better readable than what UML can offer. It also proved that users are able to work faster in AIVA when analyzing component-based applications: they find answers three times faster in AIVA than in UML.

Abstrakt

Tato dizertační práce se zabývá problémem zobrazování struktury komponentových aplikací. Použitím komponent se dá zlepšit uspořádání a čitelnost architektury aplikace díky zapouzdření implementačních detailů. Softwaroví architekti tak mohou pracovat na vyšší úrovni abstrakce, na které mohou vytvářet nové aplikace jednoduchým skládáním z menších částí. Ve výzkumu je však věnováno jen velmi málo úsilí tomu jak tyto komponentové aplikace zobrazit.

Tato práce nabízí řešení tohoto nedostatku formou nového přístupu k zobrazování nazvaného AIVA (Advanced Interactive Visualization Apprach – Pokročilý interaktivní přístup k zobrazování), který je specificky navržen pro zobrazování softwarových komponent. AIVA pracuje s dostatečným množstvím detailů, které mohou architektům pomoci porozumět všem komponentám více do hloubky. Zároveň se AIVA zaměřuje na to, jak tyto informace zobrazit bez toho, aby se zvýšila komplexita výsledného komponentového diagramu. AIVA k tomuto účelu se využívá kombinace rozdílných interaktivních technik, které umožnily snížit nepřehlednost diagramů ještě pod standardní úroveň jiných zobrazovacích přístupů. Navíc AIVA poskytuje více informací o zobrazených komponentách. Tyto interaktivní techniky zároveň zpřístupňují všechny informace co nejrychleji, což zrychluje celý učící proces.

V rámci této práce proběhla evaluace tohoto nového přístupu k zobrazování vytvořením případové studie, která se zabývá složitostí výsledných diagramů, a uživatelské studie testující rychlost tohoto přístupu. Výsledky těchto studií ukázaly, že AIVA produkuje méně komplikované diagramy s lépe čitelnou strukturou aplikace, v porovnání s tím, co může nabídnout UML. Zároveň se prokázalo, že uživatelé jsou schopní pracovat rychleji v nástroji AIVA. Při analýze komponentových aplikací nacházeli odpovědi třikrát rychleji v nástroji AIVA než při použití UML.

Contents

1 Introduction **1**

1.1 Problem Definition – Motivation 2

1.2 Aims of the Dissertation Thesis 3

1.3 Structure of the Thesis 4

2 Component-Based Software Engineering **5**

2.1 Component 7

2.2 Component Model 10

2.3 Component Framework 13

2.4 Blackbox and Other Boxes 13

3 Software Visualization **15**

3.1 Human-Computer Interaction 16

3.2 The Path of Information 17

3.3 Principles for Creation of a Mental Model 18

3.4 Visually Enabled Reasoning 19

3.5 Interactive Visualization 21

4 State of the Art of Component-Based Application Visualization Approaches **25**

4.1 Requirements of Component Visualization 26

4.2 General Visualization of Components 27

4.2.1 UML 2 27

4.2.2 Layered UML 30

4.2.3 Area of Interest in UML 31

4.2.4 SoftVision 31
4.3 Component Model Specific Visualization 33
4.3.1 SaveCCM Visualization 33
4.3.2 Palladio Visualization 34
4.4 Summary of Current Visualization Approaches 35

5 Meta-models for the Description of Component-Based Applications 37

5.1 Research Models 39
5.2 UML 2 Meta-Model 39
5.2.1 Components 40
5.2.2 Profiles 41
5.3 The ENT Meta-Model 42
5.3.1 Overview of the Meta-Model 43
5.3.2 Classification System 44
5.3.3 The Component Model Level 45
5.3.4 Application Level 48
5.3.5 Structuring Level: Category sets 53

6 Advanced Interactive Visualization Approach 54

6.1 Research Challenge 55
6.1.1 Complexity of Diagrams 55
6.1.2 Complexity of Components 56
6.1.3 Problems with CBSE 57
6.2 Conceptual Structure 57
6.3 Visual Notation of Components 59
6.3.1 Interactive Features of Component Notation 61
6.4 Diagram Filtering by Category sets 62
6.5 Inter-Component Bindings 63
6.5.1 Interactive Features to Reveal Relations 65
6.5.2 Element Oriented Connections 66
6.6 Composite Components 67

6.7 Simple Structure View 68
6.8 Conditional Highlighting 69
6.9 Other Features 71
6.10 When It Is Better To Use AIVA Than UML 72

7 Component Application Visualizer 74

7.1 Conceptual Structure 75
7.2 ComAV GUI 75
7.3 Structure of ComAV 76
7.3.1 Loader extension point 78
7.3.2 Visualization extension point 80
7.4 Supported Component Models 82
7.4.1 OSGi 82
7.4.2 EJB 3 84
7.4.3 SOFA 2 86

8 AIVA - The Implementation 88

8.1 Visual Editors 88
8.1.1 Creating New Category Sets 88
8.1.2 Condition Editor 89
8.2 Structure of the AIVA 91
8.3 JGraphBasic Plug-in 92
8.3.1 Connecting the ENT Structure with JGraphX 92
8.3.2 Node overlays 95
8.4 AIVA Plug-in 96
8.4.1 Action listeners 97

9 Evaluation of the Proposed Approach 100

9.1 Overview of Used Tools and Technologies 101
9.1.1 RSA 101
9.1.2 Technological Background 101
9.2 Case Study on Structure Readability 104

9.2.1 Case Study Approach 104
9.2.2 Results of the case study 106
9.2.3 Generalization 110
9.3 User Study on User Performance 111
9.3.1 Design of the study 111
9.3.2 Technical details of the study 113
9.3.3 Results 114
9.3.4 Discussion 118
9.4 Conclusion of Evaluation 120

10 Conclusion 121
10.1 Evaluation of Thesis Goals 121
10.2 Current State of Work 122
10.3 Future Work 123
10.3.1 Future Research Challenge 123

A Author's Publications 131

B ENT Models of Component Models 134
B.1 The OSGi Component Model 134
B.1.1 Component Types 134
B.1.2 Trait Definitions 134
B.1.3 Tag Definitions 136
B.2 The SOFA2 Component Model 138
B.2.1 Component Types 138
B.2.2 Trait Definitions 138
B.2.3 Tag Definitions 139
B.3 The EJB3 Component Model 140
B.3.1 Component Types 140
B.3.2 Trait Definitions 140
B.3.3 Tag Definitions 142

C Full Sized Images from Comparison Case Study 145

List of Figures

2.1 Component development in Visual Studio IDE 9

3.1 Diagram of visual reasoning [34] 20
3.2 A screen shot of Visual Thesaurus [71] 22
3.3 Attribute Explorer style display: (a) before changing limits and (b) after changing the lower limit [71] 23

4.1 A view on OSGi application in Eclipse dependency visualization 27
4.2 A component diagram with OSGi profile 28
4.3 An example UML class diagram of java-player taken from http://java-player.sourceforge.net 29
4.4 Multiple views in one UML diagram [15] 30
4.5 Several Areas of Interest in a component diagram [9] 31
4.6 SoftVision visualizing architecture on different layouts [54] . . 32
4.7 SaveIDE – architecture and behavioral editor [51] 34
4.8 Palladio – system editor [51] 35

5.1 MOF abstraction levels mapped on the component domain . 38
5.2 The role of common core [38] 40
5.3 Classes defined for Component diagram [39] 41
5.4 Classes defined in Profile [38] 42
5.5 Component application example — Parking Lot (OSGi application) 48
5.6 Manifest file for Gate bundle 49
5.7 The *imported_packages* trait of the *Gate* bundle in ENT representation 51

5.8 The *parkinglog* element of the *imported_packages* trait in ENT representation 51

5.9 The service *cz.zcu.kiv.parkinglot.gate* bound to bundle *Desk* in ENT representation 52

5.10 The ENT category set 53

6.1 OSGi component called "Gate" in all representation styles used by AIVA. 59

6.2 How would AIVA represent a CORBA component. 60

6.3 Two info boxes used interactively 61

6.4 Filtering ENT visualization by category sets. 62

6.5 Part of a simple OSGi application after the user requested the information. 64

6.6 Connection representation that emphasizes elements. 66

6.7 The expand principle of composite components. 67

6.8 Simple structure view over several components 69

6.9 OpenWMS application highlighted with different conditions. . 70

7.1 ComAV workspace 76

7.2 Information flow in ComAV application 78

7.3 Main menu, where new loader plug-ins are added 79

7.4 Context menu, where new visualization plug-ins are added . . 81

8.1 Editor used to create new category sets 89

8.2 Editor used to create and manage different condition rules. . 90

9.1 Workspace of RSA 102

9.2 UML profile of OSGi component model 103

9.3 ParkingLot in UML 106

9.4 ParkingLot in AIVA, with one connection (Gate – Configuration) selected 107

9.5 Complete CoCoME visualized with UML 108

9.6 Complete CoCoME visualized with AIVA 109

9.7 Comparison of average results 117

9.8 Minimum and maximum extreme values with median marked 117
9.9 Comparison of maximal AIVA results with minimal RSA results 118

C.1 ParkingLot in UML 146
C.2 ParkingLot in AIVA, with one connection (Gate – Configuration) selected 147

Chapter 1

Introduction

The overall size and complexity of current software systems is much higher than in the past. As the performance of computers rises, progressively more complex problems are algorithmized and added to software systems. For example, simple accounting systems evolved through years into highly integrated management systems offering everything from storage management or line control to enterprise resource planning and customer relationship management. This evolution of complexity is a permanent problem which did not changed from 1992, when Garlan [20] said:

> As the size and complexity of software systems increases, the design problem goes beyond the algorithms and data structures of the computation: designing and specifying the overall system structure emerges as a new kind of problem.

One of the answers to the problem of increasing complexity is Component-based software engineering (CBSE) - a new field of computer science. In CBSE software systems are composed by using components with the goal to maximize the reuse of written code, minimize the cost and the time needed for development and provide quality assurance by using certified components.

The history of components is dated back to 1968, when Mcilroy [32] firstly mentioned the concept of component. In 1987 William Atkinson, an engineer at Apple, designed HyperCard, an interactive programming tool with strong user-interface features. The concept behind this tool created the foundation for the visual component-based programming and rapid application development (RAD). It was the first time when objects became components to accelerate the development process of new software units. Microsoft adopted these principles in 1990, when it released Visual Basic, supporting both visual component-based programming and RAD, and continued in 1993 with

Component Object Model (COM) to compete with OMG's (Object Management Group) Common Object Request Broker Architecture (CORBA) from 1992. The development of new component approaches continued and Sun microsystems announced EJB (Enterprice Java Beans) in 1997 followed by OSGi component model in 1998. Microsoft also continued in their research to create a new platform called .NET in 2002, which replaced the old Visual Basic. This expansion of new component models also brought research on component models developed on universities - for example Fractal [6] in 2004, SOFA [8] in 2006, CoSi [4] in 2008 or Palladio [2]. This paragraph was based on an overview from [16].

The complexity of software systems based on any architecture is increasing in time, so in 1994 the Unified Modeling Language (UML) has taken its initial form to help software developers to specify and visualize their software. UML was finished in January 1997 as version 1.0 and supported only object oriented programming. Components were added to UML in version 1.1 (fall 1997) but they were used to represent implementation items, such as files and executables, which is in conflict with the common use of the term "component". These conflicts were resolved in version 2.0 starting in July 2005, when UML officially changed the meaning of the component in its diagrams. Since then the UML is the mainstream approach in visualization of component-based applications.

1.1 Problem Definition – Motivation

The motivation to study structure of component-based systems can differ, but generally we can state two common scenarios: 1. Company overtakes the software project of another company, to continue in development, or to add new functionality; 2. New team member is hired to work on project in progress. In any such case it is vital to understand the structure in order to work with the system.

Visualization is important in order to gain insight, to understand the structure of software system and to enable decision making. This statement can be challenged, because by studying implementation itself one can also gain knowledge of a software system, but we claim that this is rather hard in the context of large software systems (e.g. over 100 components) and that visualization is a great help in the process of learning such structures.

The structure of large software component-based systems is very complex and difficult to visualize all at once, when one also needs to work with some details. The result is usually rather hard to read, as details mix with structure and obscure the clarity of the diagram. Components can be very different and have different features, so it is also very hard to design a general way to express these details.

There are several roles in every component-based development approach, which are interested in different information. For example, component developers are interested in all details to be able to create new components and connect them with others, while component assemblers are only interested in relations between components so they can compose new systems without information overload. This different information can even be on different levels of details. The problem then is, how to visualize multiple levels of details and how to filter provided information to enable different roles to read a visualized diagram with the information wanted. The common practice is the creation of separate diagrams for every need, which unfortunately brings new problems with the sustainability of several diagrams and higher cost needed to maintain these diagrams.

Interactive techniques are known from other fields of information visualization and these techniques introduce ways to overcome these problems – the need for several diagrams and poor readability of the whole models. Some interactive techniques can also increase the speed of the learning process itself. The problem is that there is no approach that uses principles of interactive visualization for the purposes of component applications structure visualization.

There are approaches that visualize structure and details of component-based applications, but they do not use these interactive techniques. There are also approaches that use interactive techniques, but they are rather focused on analysis and a general overview of a visualized system and can't provide the required details.

1.2 Aims of the Dissertation Thesis

From the previous discussion it is obvious that the problem is in the absence of an approach that would accommodate interactive principles and adopt them in structure visualization with a sufficient amount of detail. The main goal of this work is to design a new visualization approach, which should be able to:

1. Visualize the structure of any component-based application as a graph diagram.
2. Visualize a sufficient amount of detail.
3. Provide ways to filter these details and work on different levels of detail interactively.
4. Maximize the advantages of interaction to boost the learning process.

Prior to visualization, it is necessary to develop a data structure that is able to hold information about any component-based application with a sufficient amount of detail. The ENT meta-model [3] provides a detailed and general description of single components. This meta-model was developed during previous research in our group, so we decided to reuse it and extend it.

1.3 Structure of the Thesis

The structure of this thesis is as follows: Chapter 2 covers the basic terms of component-based development and discusses the issues of very different understandings of components. Chapter 3 then provides an introduction to visualization with a focus on the cognitive limits of the human brain and how interaction can be helpful in the process of learning a software system. The state of the art of component-based application visualization approaches is covered in Chapter 4, with a deeper description of problems of current approaches. Chapter 5 provides an overview of current meta-models used for the description of component-based applications. A special care is given to the used ENT meta-model, which is thoroughly described together with its formal specification.

The proposed visualization approach, called AIVA (Advanced Interactive Visualization Approach), is presented in Chapter 6. All aspects of this new approach are covered in this chapter from theoretical point of view. Next Chapter 7 describes the implementation of the ComAV visualization and reverse-engineering platform, covering the reasons why it was developed. Chapter 8 will finally reveal the implementation of the AIVA as a plug-in for the ComAV. This new approach is then evaluated using case-study and user-study, these studies are introduced together with results in Chapter 9. Finally, this whole thesis is concluded in Chapter 10.

Chapter 2

Component-Based Software Engineering

Component-Based Software Engineering (CBSE) is a branch of software engineering that emphasizes modularity and extendability by composing whole software systems from separate building blocks called components, which communicates through interfaces. The principles of CBSE are described in a great detail by Heineman [22] and Crnkovic [11], [12]. The idea of components is taken from other industries where this concept is well known and has been used for many years. Components are prefabricated "things" that can be rearranged to create new composites that are required by a customer. This principle transferred to the development of new software systems has three steps, performed by a software architect in the initial phase of a project: the software system is divided into separate abstract components (how components should look ideally); the component repository is searched for real components that can satisfy the needs of the abstract components; when no real component can offer the required functionality, a new component has to be designed, implemented by component developer and added to the component repository.

A component should provide integrated functionality for one problem or a group of similar problems to enable future reuse of itself, and it is designed with this purpose in mind. This might be a very difficult task, because if component should be reusable, it has to be general, but still it has to avoid over-generalization in order to provide enough functionality that reuse remains practical. As mentioned before, components are stored in component repositories from which assemblers can take finished components and use them to create a final product. Every software company has its own component repository where they store their own components together with purchased ones. Purchase of finished components is practical, because it is much cheaper than the development itself, but on the other hand, there is no official certificate of component quality. Although there is no official certification of components, the biggest reseller of software components, componentsource.com (over one million members and two thousand components), offers customer reviews to provide at least a limited idea about the component quality.

We would like to define basic terms in the field of CBSE to provide a foundation for the rest of the paper. Simple, but comprehensive definitions are provided here, while more precise descriptions will be provided in subsequent subsections. In CBSE there are three elementary terms:

- **Component** - a unit of composition or extension.
- **Component model** - a description of a component and the component's environment.
- **Component framework** - an environment in which components are deployed.

At this point, a more formal definition of CBSE can be quoted from Bachmann [1]:

> Component-based software engineering is concerned with the *rapid assembly* of systems from components where: components and frameworks have certified properties; and these *certified properties* provide the basis for *predicting the properties of systems* built from components.

or a different one from Heineman [22]:

> The major goals of CBSE are the provision of support for the development of systems as assemblies of components, the development of components as reusable entities, and the maintenance and upgrading of systems by customizing and replacing their components.

The above statement from Bachmann mentions the biggest motivation for use of components, which is rapid assembly, resulting in overall rapid development. Software systems can be assembled rapidly by using components from a repository whose components were developed or bought earlier; thus it saves time and resources and reduces the price of the final product. With a sufficiently comprehensive repository of components, this approach can overcome the well known paradigm: *cost, time, quality; pick any two.* In the repository, it is logical to maintain only components that have in some way certified features and properties (on a company level) and by using these components it is predictable how the final system will behave and what the properties of a system will be. This also guarantees some level of quality, because by using quality components and by predicting the properties of the final system, it is possible to rapidly assemble a final system that has quality attributes adequate to the components used.

Any component depends only on its requirements, which are well-defined and have to be satisfied. One of the consequences is that a component is independent of other components. When an application should be extended, it does not include changes of old components, but adding new ones and satisfying their needs. This means that components that compose the application are independent of newly-added components – resulting in independence for extensions.

At the end of this introduction, we would like to summarize the motivation for using components:

1. Components can be bought on component markets to save time and costs.
2. Time-to-market can be reduced by using components.
3. The quality of the whole system can be predicted by using certified components.
4. Component systems can be independently extended.

2.1 Component

The term "component" was already mentioned earlier, but what, in fact, a component is, is still unexplained. This is caused by the very broad understanding of what a component is. Before we continue in theory, lets look at a few specific component descriptions that will help to understand the term.

- **OSGi component** [40] is called a bundle and it is deployed directly into an OSGi framework. In the framework, all the components are equal and ready to provide the services they offer. A service is an interface that describes what is provided outside the component for use by other bundles. Bundles can ask the framework to provide them with a service conforming to the requested interface. All the communication between components is realized through these services; thus, applications are composed by simply deploying all components that make up the application into the framework and the framework manages the rest. Physically the bundle is written in Java language and distributed as a jar file with an extended manifest, to be able to describe the bundle more precisely (what it needs and provides *inter alia*). In this jar file there is a main class file, which contains implementation for the starting and stopping sequences of the bundle. The context of the framework is handled to this class, so it is able to register new services or receive previously registered ones. A service is an instance of a class, that implements interface of the concrete service.

- **SOFA component** [8] is called architecture and its description is called a frame; which says which interfaces are provided and required by the frame. SOFA is closely bound to the component repository from which components are taken and deployed in one or more nodes; these nodes are distributed frameworks that manage the life cycle of a component. SOFA is an implementation of a hierarchical model; thus the application, called an assembly, contains a pointer on the top level architecture, which is a component composed of several subcomponents. A component which is composed of several subcomponents is a composite component, in a general terminology. A composite component can make use of its subcomponents and these can communicate between themselves. SOFA is also based on Java; thus the implementation is Java classes. Physically, the architecture is only an XML description that says how the component is composed, what frame it implements and which Java class is the implementation. SOFA components are always stored in the repository together with XML descriptions of frames and assemblies. When an application should be started, a user has to select a deployment plan, where is described which assembly should be started on which nodes.

- **.NET visual component** [1] is used in Visual Studio IDE to compose Windows applications. These components are composed from other visual subcomponents and are referred to as "UserControl". The application logic of the component is written in any .NET language (C#, Visual Basic or any other) and it has to be built, before first use. These components are closely bound to IDE and are integrated into it, unlike OSGi and SOFA. The primary objective of user controls is to be used as GUI elements with application logic attached to it. One or more user controls can be built and distributed as a DLL library to a third party. This DLL has to be added to references before it is shown in the toolbox of IDE and can be used. This can be confusing, so it is better to use an image (see Figure 2.1) to illustrate the usage of user controls. In the top left corner you can see the toolbox of user controls,

[1]MSDN library about user controls http://msdn.microsoft.com/en-us/library/y6wb1a0e.aspx

which can be drag-&-dropped to build new user controls. In he right part of the image, you can see a selected user control, which contains a list box with fruit names and a button, that is able to add new fruit in the list box.

User controls are physically composed of two files – a class file that contains application logic; a designer file that contains information about graphic elements. A parent component can use any property or method that is set as public in the class file; moreover, user controls provide events, which are callback methods used by a parent component to react to an event triggered by its child component. User controls are not deployed into the .NET framework, but they are integrated into the application in the building process; thus the application is built as monolithic.

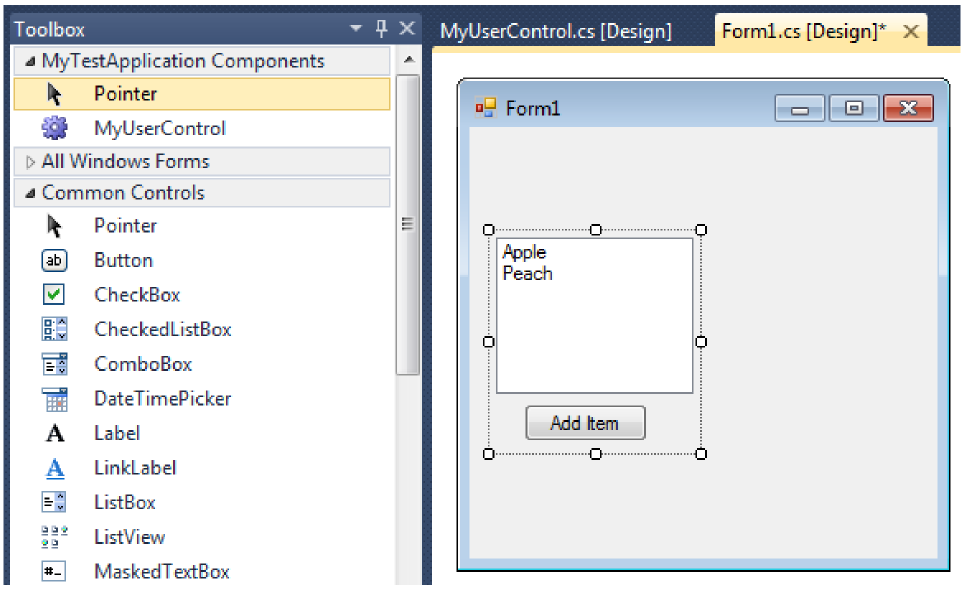


Figure 2.1: Component development in Visual Studio IDE

All the above-mentioned components have very different structure and usage, which is common for different components. Now we have to highlight that by different components we mean components conforming to a different component model, because there is no component without a component model. In other words, a component has to conform to a component model, because otherwise we can't speak about a component at all.

There are a few things that components have in common, which will be clear from definitions provided below. Bachmann defines a component as follows [1]:

> A Component is:
>
> - an opaque implementation of functionality
> - subject to third-party composition
> - conformant with a component model

Szyperski defined a component in [64] differently:

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

Yet another definition from Taylor [65], which is more focused on architecture:

> A software component is an architectural entity that (1) encapsulates a subset of the system's functionality and/or data, (2) restricts access to that subset via an explicitly defined interface, and (3) has explicitly defined dependencies on its required execution context.

These definitions should be discussed to provide proper explanation. It is clear that components have to implement some functionality, because otherwise it would be impossible to create any application using them. This implementation is, however, hidden – encapsulated inside a component. A component can be used only through explicitly defined interfaces – meaning we can use functionality without the need to know how it is implemented. An explicit definition of interfaces results in well-defined interfaces and enables a third party to use the component, without any other knowledge. Such components can be then composed together by a third party. Components can depend on some resources – files, classes, framework services or other components. However, these dependencies have to be also explicitly defined, so a third party can satisfy these dependencies. A component has to conform to a component model, otherwise it cannot be either composed or deployed. And finally, a component can be deployed independently of other components, because dependencies are resolved after deployment.

2.2 Component Model

A component model is, as mentioned earlier, a description of how a component should look, interact and be deployed. We also mentioned that a component that does not conform to any component model is irrelevant, because two components can interact if and only if they can create assumptions about the other component, for example, how to locate the second component, how control flow is synchronized, which communication protocol is used, how data is encoded and so forth. In this subsection we will discuss what must be described by a component model in order to use it. Lau provides a simple but elegant definition in [30].

> A software component model is a definition of
>
> - the semantics of components, that is, what components are meant to be,
> - the syntax of components, that is, how they are defined, constructed, and represented, and
> - the composition of components, that is, how they are composed or assembled.

There is no agreement on what should be described by a component model, but based on our study of Bachmann [1], Lau [30] and Szyperski [64], we found five important things that are commonly covered by component models. An exhaustive list of things that a component model describes is given in [13], where Crnkovic defines the classification framework for component models.

Component types. In the above definition it is mentioned, that a component model has to define the semantics and syntax of components. But it is possible that there are different types of components in the sense of different building blocks. Some component models can recognize more than one component type, where every component type has its special purpose – for example, EJB 3 (Enterprise Java Beans) [63] has three component types, SessionBean contains application logic, MessageDrivenBean can listen to events and Entities are used as DAO (Data Access Object). In such cases, the component type acts as an interlayer between a component model and the specification of semantics and syntax – a component then has to conform to the component type to be recognized by the component model. Every component model recognizes at least one component type. By introduction of component types, the above definitions remain valid, and it is the purpose of component types to provide the semantics and syntax of components.

The semantics of a component define how the component should look – what its purpose is, how it can communicate, what it can provide and require, how it is deployed, etc.

The syntax of a component defines how the component should be implemented – required files that have to be present in every component with the description of these files, where the source code is located, implementation requirements, which interfaces have to be implemented, etc.

For example, JavaBeans and EJB's SessionBeans are both syntactically Java classes, however, different semantically. JavaBeans are hosted by a container and interact with one another via adapter classes generated by the container that link beans via events. SessionBeans are hosted and managed by an EJB container (different type) and interact with one another via methods that are provided by two interfaces – home and remote.

Interaction schemes. When components are deployed, they have to be able to communicate between themselves in order to create a functional unit. The component model may describe how components interact with each other, or how they interact with the component framework. There can be restrictions on which component types can communicate together and which can not. The interaction itself can be realized in very different ways – through network communication, interface calls, pipes, events, intermediates, etc. The interaction also includes things related to resource management, thread management, persistence and so forth.

Architecture styles. The software architecture is very important in CBSE, because it can affect not only how the system should be built, but also some quality attributes; systems with better architecture can have a better response. The component model can prescribe architecture styles that are allowed – how components are composed and which component types can be composed together.

Resource binding. Resources in the scope of component models can refer to files, classes, services provided by the framework or other components. A component can use one or more resources provided either by the framework or by another component. A component model describes which resources are available to components, and how and when components bind to these resources.

Deployment process. A component model can also describe how components are deployed into the component framework. In OSGi, all components have to be installed into the framework prior to being started; in SOFA, it is only necessary to start the deployment plan, because all the components are automatically taken from the repository and distributed into the deployment nodes; and, finally, .NET visual components are automatically integrated into the application in the building process and run monolithically.

2.3 Component Framework

A component framework is an implementation of a component model that enables components to be deployed and run. A component framework manages resources shared by components, components themselves, communication between components and the whole life-cycle of components. A component framework has to enable exactly what is described by the component model. Bachmann [1] recognizes two types of component frameworks:

1. **Runtime framework**. This type of framework offers an environment for components where they can be deployed and creates a layer between components and the operating system. Components are after that managed similarly as processes in the operating system: they can be started, suspended, resumed or stopped. Compared to operating systems, frameworks offer only limited interaction mechanisms equal to the ones described in the component model. The OSGi and SOFA component models use this type of framework.

2. **Bundled framework**. In some cases it is not suitable or necessary to work with components one by one, so the framework is bundled with components and behaves more like the bottom layer of the application, which offers services and abstraction from the operating system. This type of framework doesn't manage the life-cycle of components.

2.4 Blackbox and Other Boxes

A blackbox is a device that has a well-defined input and output and no internally observable state. A blackbox can be used without knowledge of how it works; one only needs to know its description. In software component analogy, we say that a blackbox is a component that can be used solely by knowing its interfaces – required and provided ones – and without knowing implementation details. In other words, a blackbox can be reused by a third party without relying on anything but its interfaces and specifications. The blackbox nature of components is a very important principle, based on information hiding, as discussed by Parnas [43], who said that modules should hide their internals and make only selected features accessible through its public interface. Brada [5] also discusses the importance of a blackbox in CBSE.

There are also other patterns that differ in the opacity of implementation. The opposite to a blackbox is a whitebox, which allows the user to study implementation details to enhance understanding of the component. A whitebox can be reused through its interfaces, but it relies on the understanding gained from studying the implementation details. Some authors even suggest the usage of a glassbox, allowing only a study of the implementation, while a whitebox allow even manipulation with implementation itself. The use of whiteboxes can be dangerous; this fact is presented by Szyperski in [64].

> Whitebox reuse renders it unlikely that the reused software can be replaced by a new release. Such a replacement will probably break some of the reusing clients, as these depend on implementation details that may have changed in the new release.

In the middle of these two patterns are grayboxes, which reveal only a controlled part of their implementation to enhance the understanding of the component. Buchi, for example, claims that components should be grayboxes and presents evidence in [7]. Grayboxes are used, for example, in the SOFA component model [8] in the architecture of frames.

Chapter 3

Software Visualization

The discipline of software visualization is introduced in this section, together with basic cognitive and psychological principles applicable to the field of software visualization.

Visualization is the name of a discipline of computer science that is interested in transformation of information into visual form, in order to help scientists and engineers see otherwise hidden features. There are two major disciplines of visualization: *scientific visualization* processes physical data, whereas *information visualization* processes abstract data. Software visualization is a part of information visualization, because programs and algorithms do not have physical form. Software visualization is concerned with visualization of applications or parts of applications from different points of view. A formal definition of software visualization was given by von Mayrhauser [68]:

> Software visualization is a discipline that makes use of various forms of imagery to provide insight and understanding and to reduce the complexity of the existing software system under consideration.

As outlined by this definition, one can imagine that the discipline of software visualization is an extensive field of study. Diehl recognizes three categories of software visualization in [14]:

- **Structure** visualizations can visualize internal static qualities of software; these qualities can be inspected *without running the program*, as they are based solely on the implementation itself. This type of visualization is supposed to help to model the architecture of software, describe classes, present algorithms in a visual way[1], etc.

[1]One can object that algorithm is dynamic, but it is still something that can be inspected just by analyzing the implementation.

- **Behavior** visualizations can visualize dynamic qualities – function calls, memory usage, run-time, etc; these qualities can only be inspected *after running the program* and analyzing how this program behaved. This visualization is supposed to help with finding slow or resource-draining parts of an application, visualize the sequence of function calls, etc.

- **Evolution** visualizations can visualize both static and dynamic qualities, but it emphasizes how these qualities change in time; e.g., it visualize the changes of source code.

These categories can be challenged and new ones can be designated, but we provide them just to present how different things can be part of software visualization and not to discuss them in detail. There are a lot of things that can be said about software visualization and Diehl provides an exhaustive number of details in [14], Taylor provides a more brief, but also very interesting description of software visualization in [65].

We are not the the only ones who want to address problems of current software visualization approaches. Knight [27] discusses the problems related to the comprehension of programs and suggests the use of three dimensions. However, there has not been any revolution of 3D software visualization approaches after nine years, so we approach very similar problems to those that Knight described, but by concentrating on interactive techniques.

In the scope of this thesis, we would like to talk more about the ideas behind visualization that will back up our proposal, so we will not continue with a description of software visualization itself. Instead, we gave more space to interaction and psychological principles because if one wants to design a new visualization approach he has to understand to the cognitive limits. Following sections will therefore cover basics of human computer interaction in Section 3.1, what is needed to be done to transform a data into insight in Section 3.2, basic visualization laws based on cognitive limits in Section 3.3 and finally how important is interaction for visualization and an overview of interaction techniques in Sections 3.4 and 3.5.

3.1 Human-Computer Interaction

This section will briefly introduce Human Computer Interaction (HCI) and its impact on software visualization. A basic goal of HCI is to improve the interaction between user and computer. A long term goal of HCI is to minimize the barrier between the human's cognitive model and computer's understanding of user's task. HCI is concerned with problem of making computers more usable and receptive to the one's needs. HCI itself is a mixed field of computer science, human psychology and several other fields.

Software visualization is generally a field that can benefit much from different HCI methods. No software visualization approach is able to visualize a complex applications without the need to interact; at least some kind of navigation is always included. Software visualization tool with better HCI can provide better understanding and reaction on one's needs. Therefore, different HCI can improve the overall capabilities of the visualization – if one will get what he expects, he can gather the required knowledge easier.

3.2 The Path of Information

The software visualization itself is only one step on the path of information. There is a raw data on the start of this path and insight on this data at the end. This section will shortly elaborate each step, which should provide motivation why is visualization so important. It should also help to distinguish between model and visualization. These steps are defined based on our experience and summarize the information given in the rest of this chapter.

1. **Raw data** are the input. It is typically a source code or configuration files of an application that should be understood in order to work with this application.

2. **Information model** is an abstraction of reality. An abstraction of source code can be model that loses implementation details, but keeps all classes, methods and public properties. Such model is understandable to the computer that is therefore able to process it further. The format of this model must be defined by a meta-model. An information model of software can be either created manually or using reverse-engineering.

3. **Visualization** is a visual representation of an information model. The purpose of software visualization is to move an information model from computer to mind. It also helps to refine already existing mental model.

4. **Mental model** is an abstraction of reality in mind. It is mandatory to analyze the data and to gain insight. Mental model is described in detail in the next section.

5. **Insight** is knowledge and ability to reason.

This will be better explained on an example. There is a Java application and one needs to understand it. It's source code is a raw data and it will be studied thoroughly later. One will use some CASE (Computer Aided Software Engineering) tool to reverse-engineer the structure of the Java aplication, therefore it will create an information model. CASE tool will probably use UML to visualize this model which can be used to walk through the UML diagram. After a while one will learn the application's structure and he will create a mental model for it. Once this mental model is created one will get an understanding to this application – an insight and he will be able to make decisions regarding this application.

3.3 Principles for Creation of a Mental Model

Before the analysis of a visualized software can start, it is of most importance to create a mental model. A mental model is a representation of reality in mind, which is used in the thought process. Human reasoning depends upon a mental model, which can be constructed from perception, imagination, or the comprehension of discourse. This established theory is developed and described by Philip Johnson-Laird [26].

Ric Holt uses the theory of mental models and he applies it to software architectures in [24], where he defines basic cognitive principles applicable to software architectures, which facilitate creation of a mental model of a software system. These rules are valid also for visualization of components, and have one thing in common – they try to minimize what is learned, to avoid a brain overload caused by complexity. Holt identified several laws and principles, from which we will discuss three laws that we see as most important for the visualization of structure:

- **Law of maximal ignorance.** *Don't learn more than you need to get the job done.* When visualizing large and complex component systems, one must filter away unwanted detail to promote simplicity. It is often advantageous to oversimplify the representation of the implementation, to make it easier to think about the architecture, but one has to realize the danger of these simplifications. When studying the architecture of complex component systems, it is important to keep one from learning too much, because details cause distraction and extend the time needed for the creation of a mental model, thus extending the time needed for reasoning about this mental model. However, too much simplification may conceal necessary information.
- **Law of minimal change.** *When the software changes in a modest way, our model for it should also change in a minimal way.* When

visualizing two versions of the same component system, it is important to keep the models similar – the same layout with components and clusters of components positioned at roughly the same place with the same colors (if any were used). Each visual change in a represented system means a change of the mental model for every team member. These changes are time-consuming, cause confusion and are error prone.

- **Law of ugliness hiding.** *Unobserved ugly parts of a system stay ugly.* When ugly parts of system are hidden, they can't be recognized and repaired. People instinctively like things to be clean and simple, so when they see something messy that ought to be simplified, they tend to fix it. This law should emphasize, that the resulting visual representation should be complete and should not omit ugly parts or they may not be fixed.

One last thing about the brain is, that visual information (shape, color, texture, position) is processed in the right hemisphere and verbal information (text, spoken sentence) is processed in the left hemisphere, so when both these types of information are used together, we can use both our hemispheres to create a mental model of the represented system. Therefore, a combination of both these types of information sources would provide better results. It is commonly referred as the dual-coding theory, described by Paivio in [42].

3.4 Visually Enabled Reasoning

Meyer et al. define the new science of visually enabled reasoning in [34] which evolved from visual analytics by concentrating on interaction and interactive reasoning. We would like to present some basic ideas to demonstrate that interaction should also be part of software visualization, because it enables us to work faster and more efficiently and helps to create a mental model of a component system in order to gain insight and enable decision making about that system. The importance of interaction to a gain of knowledge or insight is also depicted in Figure 3.1. One can gain some level of knowledge from either pure data, analyzed model or from visualization by interacting with one or the other. However, only interaction with visualization can provide visual reasoning, which is most important in order to gain the insight.

Visual Analytics itself is a field of information visualization that incorporates HCI with respect to data analysis. Visual analytics facilitates data analysis through HCI. The human visual and cognitive systems are the most powerful tools for understanding complex relations, so in order to maximize

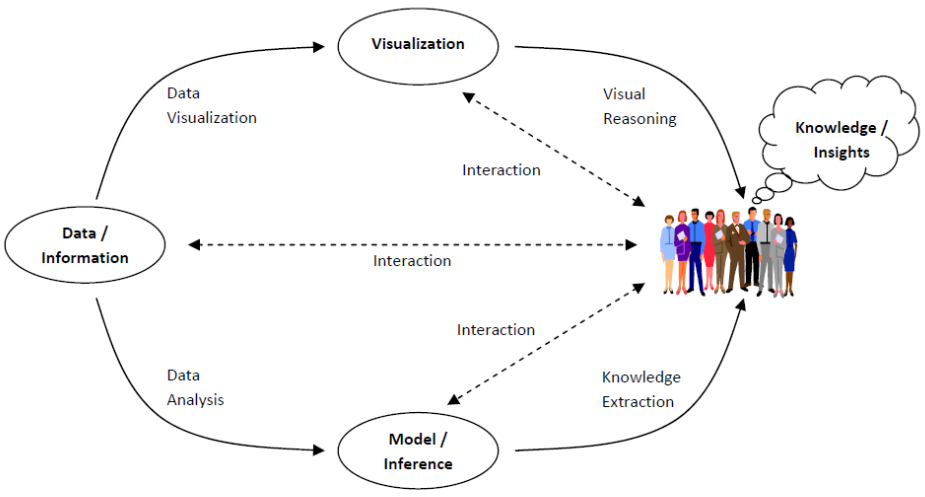


Figure 3.1: Diagram of visual reasoning [34]

user experience and performance it is essential to use the advantages of dynamic interactive principles and adapt them to create a perfect match with our visual and cognitive systems. A system that matches visual perception, with respect to resolution, focus, attention and detail without overloading human senses is most suitable for efficient interpretation of large data sets [34].

Interactive Reasoning is the process of distinguishing between ideas in order to create new relations and insights based on collected evidence [34]. Evidence can be freshly gathered from a visualized representation or based on previous knowledge. Evidence can be any information, data, idea or artifact resulting from reasoning. Interactive visualization is not only what is visualized, but also how – user interface, interaction with user, manipulation of the visualized representation. These elements of HCI should offer the user enough means to make progress in the reasoning process.

Insight Gain is the ultimate goal of visualization, because insight involves knowledge and the ability to reason about a mental model.

Meyer et al. also mention that a big shortcoming of current interactive visualization systems is that they depend only on the visual sense. This is caused by the fact that the visual sense comprises as much as 75% of all information perceived from the outer world. Meyer suggest, that involvement of other senses could enhance the possibilities of interactive visualisation even further.

3.5 Interactive Visualization

We already discussed HCI in Section 3.1and its importance for information visualization and visually enabled reasoning in previous Section 3.4. Interactive technique is a mechanism for modifying what the users see and how they see it. There are a lot of interaction techniques that provides this interaction within the data and information visualization context. This section will provide their categorization to provide more understanding to this problematics. These categories are the result of study by Yi et al. presented in [71] and can be considered as key for our future work, because they offer different interaction categories based on user intent.

The recognition of this categorization may be seen by using it by Ward et. al. [69] in his book. Inter alia, Ward et. al. look on interactive techniques and describe them in terms of operators and the operand. Interaction operator is the core of the interactive techniques itself, interaction operand then describe the space upon which the operator is applied. This description (operator, operand) should help to define an architecture that combine different interactive operators with different operands to design better interactive architecture.

The categories of interaction techniques are the following:

Select: *mark something as interesting.* This enables users to select the items of interest, which are highlighted in some way to keep track of them. This is extremely useful when too many data items are presented all at once, or when the representation of a system is changed, for example, when changing the layout of the system. By marking selected items in a sufficiently distinctive way, it is easy for users to stay oriented even in large systems or in a dynamically changing environment.

Explore: *show me something else.* In a more complex system, it is not possible to visualize all the items at once, because of screen resolution and cognitive limitations. To overcome this limitation, it is important to enable the exploration of the system. By exploration we mean moving from one point of interest to another in order to gain understanding or insight of the whole system. This exploration can be achieved by simple scrollbars that enable moving over the big diagram, while visualizing only a small part of it. On a very similar principle, panning also works, enabling one to drag a canvas and move it while the camera is steady. Other approaches can offer smooth transfers from one point of interest to another on one click, or even rearranging the view, based on the actual point of interest. All these techniques share the goal of the exploration of a system in order to gain understanding and insight.

An example of an explore technique is a “Direct walk”, used in Visual Thesaurus. Searched vocabulary is shown in the center surrounded by related vocabularies. When one of the surrounding words is clicked, it will smoothly become a new center of the screen. Figure 3.2 shows such related vocabularies for the word get.

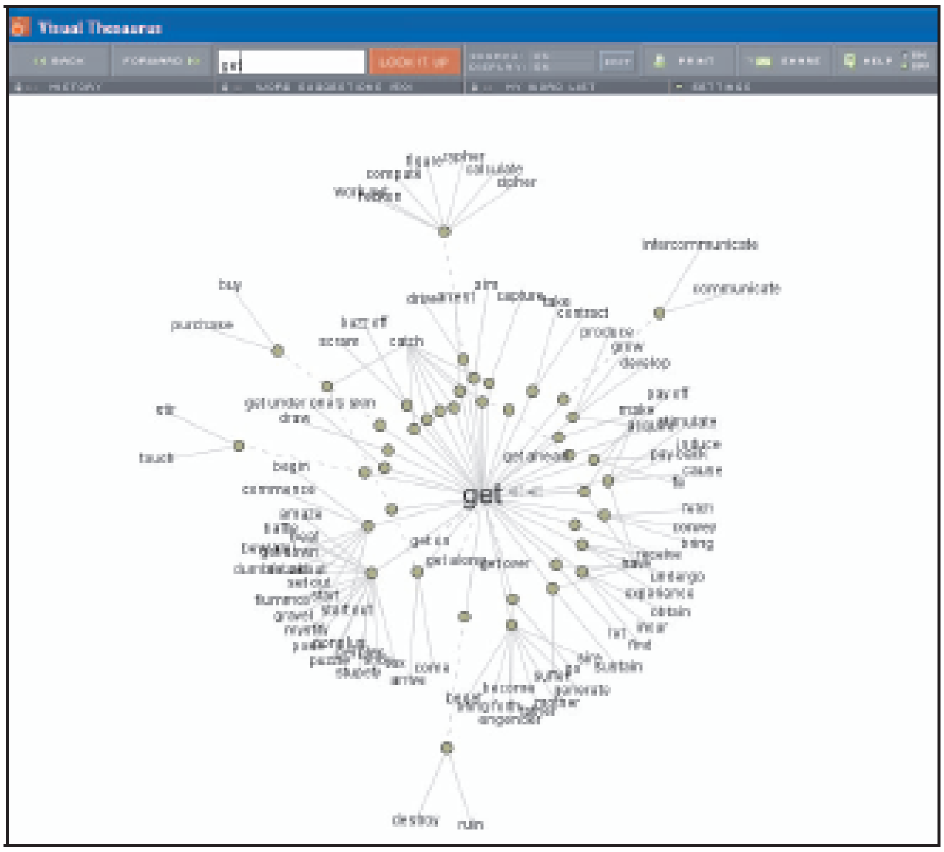


Figure 3.2: A screen shot of Visual Thesaurus [71]

Reconfigure: *show me a different arrangement.* Every visual representation of a system has its own spatial arrangement of the items – layouts. Every layout is made with a purpose in mind, to emphasize some hidden characteristic of the system. Layouts can emphasize relations – e.g., hierarchic relations arranged as a tree; similar characteristics of the items – e.g., items with similar characteristics can be clustered together; or any other, depending on the need. The important thing is, that to reveal the real nature of a complex system, it is beneficial to change the layout in order to gain a different perspective. The reconfigure category includes all techniques that can help to rearrange the spatial representation of the items in order to reveal hidden characteristics of the represented system, but we think that for software visualization in 2D diagrams, the layout switching is the most important technique.

Encode: *show me a different representation.* Techniques from this category change the visual representation of the items – color, shape, font, size, etc.. These changes are made in order to add or emphasize some characteristics of the items. In software visualization we can change the representation of components or lines that connect components – lines can be collapsed or separated, a component can be represented in UML style (box, with text information) or as houses [70]. This technique provides another view of a component. Another widely used technique is to change the color, based on a certain variable. These colors can mark the components with different characteristics, e.g. response time – green for fast response, orange for medium response and red for slow response. This approach emphasizes some feature of a component.

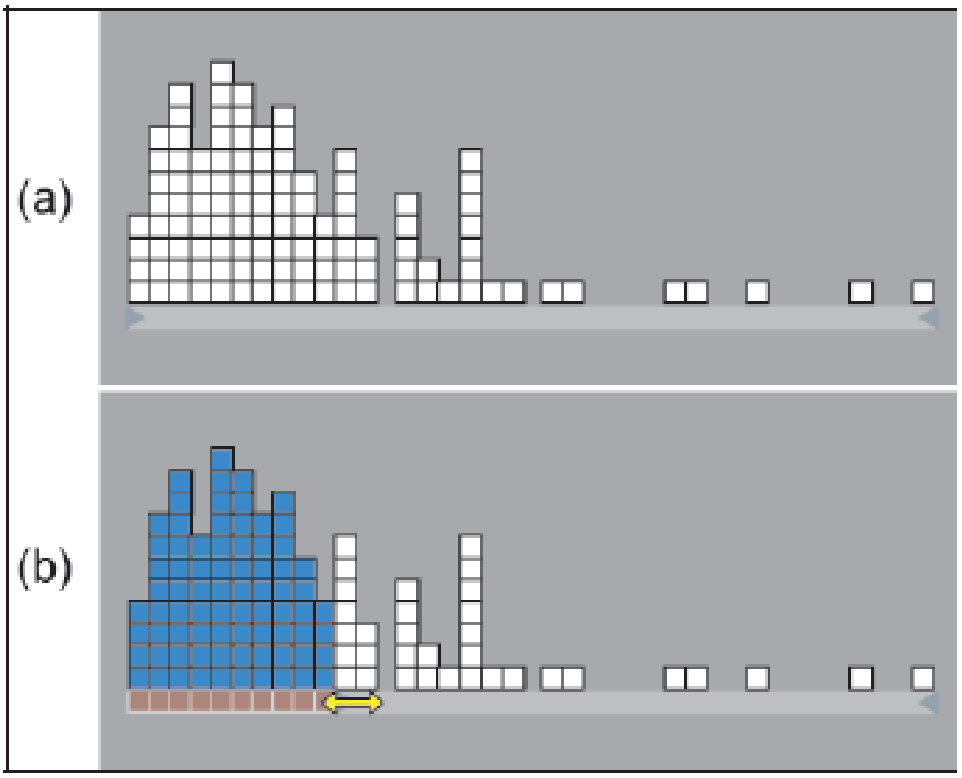


Figure 3.3: Attribute Explorer style display: (a) before changing limits and (b) after changing the lower limit [71]

Abstract/Elaborate: *show me less or more detail.* These types of interaction allow users to change the level of detail from an overview to a detailed study of individual attributes. All types of the details-on-demand technique are in this category. *Lens* is a technique that works as a magnifier; it does not simply magnify the hovered part of a diagram, but shows details instead. *Tooltip* is a technique that shows details after hovering over a data item. *Drill-down* is a technique that shows the internal structure of hierarchical components, but revealing it only if this hierarchic component is clicked. Along with details-on-demand techniques, we can also refer to contextual zooming, which changes the level of details based on distance: when zoomed out we see only boxes-and-lines, when zoomed close enough to be able to read, component elements are revealed.

Filter: *show me something conditionally.* These techniques offer functions that hide or show differently the items that do not match the criteria. These techniques aim to filter unwanted detail interactively with the possibility to cancel the filter or change the filter – e.g., the hidden items can be shown again. These techniques can filter out components, elements of components or connection lines. The difference is in the way these items are filtered – e.g., they can be hidden, marked with a color or blurred with depth of field. An example of how to filter elements using different colors is in Figure 3.3.

Connect: *show me related items.* A user who needs to reveal the relations between components will use techniques from this category, because they highlight associations and relations of selected items. – e.g., highlight all components directly connected to the selected one. This technique appears in many visual forms – e.g., edges are made bold, shades of components are colored, unrelated components are blurred with depth of field. Another technique goes across categories, because it hides all unrelated items, shows all related items and arranges the selected item in the center of the screen; all related items and only these items are then arranged around this selected item and when a new item is selected, the whole process repeats.

Chapter 4

State of the Art of Component-Based Application Visualization Approaches

CBSE is now a mature field of study, with dozens of component models like EJB [63], CORBA [35] and OSGi [40]. More can be found in commercial applications and even more component models – for example, SOFA [8], Fractal [33] and CoSi [4] – are the subject of research. Every component model can describe a component in its own way and introduce some special features of these components, for example, behavior or interaction.

In such an environment, where component models have so little in common and can have so many different characteristic features, component architects and assemblers are forced with these choices of how to visualize the structure of their component-based applications:

1. Use a general "boxes-and-arrows" visualization;
2. Create a component model-specific visualization.

Neither of these two choices can provide a solution for all the problems stated in the Introduction, but as it will be shown in the first subsection there is a way to instantiate a general visualization approach for the purposes of a concrete component model, thus providing a sufficient amount of detail. However, due to generality of this approach, it is still impossible to provide advantages that a component model specific visualization can offer, which will be discussed in the next subsection. Any general visualization able to visualize details has to be built on top of a good meta-model that is able to provide this generality together with details – this will be subject of the last subsection.

4.1 Requirements of Component Visualization

The field of CBSE was briefly discussed in Chapter 2 to provide an overview of basic terms. However, it was not discussed who is involved in development of component-based applications – who will use visualization tools that display these applications. Hence, it will be discussed now. Basically, there must be at least three different roles: system architect, component developer and component assembler. Moreover, Szyperski [64] defined a framework architect role, however we think this role is not so important for software development thus we will not discuss it further.

Component developer works on the lowest abstraction level – he develops components, he makes the black-boxes to work. On such level he needs as much details as he can obtain about other components that will be connected to the component he is developing. He simply needs to know every single detail to develop component that can be easily assembled with other components.

System architect is the person who designs a system as a functional unit. He decide how should components look, what they should do, etc. System architect needs access to almost the same amount of detail as a component developer need while he still needs to keep an overview of the whole system unit. This role would benefit most from approach that could provide it an instant transition between structure view and detailed view.

Finally, component assembler composes the final system. He takes functional components and put them together. He needs to know what every component needs and what it provides. He must be sure that these components will work together and he must make sure the requirements of all components in the system are satisfied. Component assembler needs the least details, he is only interested in component's needs that he has to satisfy. He works on the highest abstraction level, generally he is mainly interested in the structure of the system unit. However, he still needs to access some details, while others are only disruptive for him.

A visualization approach should consider needs of these different roles in order to provide a useful visualization tool. More detailed view on needs of these roles and what could visualization tools offer is in our publication [25].

4.2 General Visualization of Components

A general "boxes-and-arrows" visualization is useful for the exchange of diagrams between domain experts, but it provides only a few specific details about components and thus it can only provide a shallow understanding of the component-based application. The Eclipse dependency visualization[1] is a great example of general "boxes-and-arrows" visualizations. A simple example is in Figure 4.1.

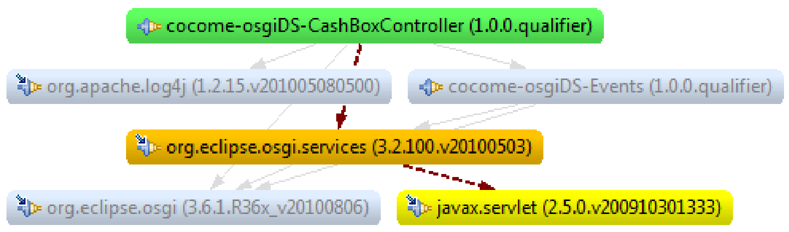

cocome-osgiDS-CashBoxController (1.0.0.qualifier)
org.apache.log4j (1.2.15.v201005080500)
cocome-osgiDS-Events (1.0.0.qualifier)
org.eclipse.osgi.services (3.2.100.v20100503)
org.eclipse.osgi (3.6.1.R36x_v20100806)
javax.servlet (2.5.0.v200910301333)

Figure 4.1: A view on OSGi application in Eclipse dependency visualization

Because these general approaches do not introduce any ideas interesting for component visualization, we will not include them in this overview. A more sophisticated example might be the UML 2 [39] component diagram, which can show more information about single component; thus it is not only "boxes-and-arrows".

The answer for how to visualize a component-based application in a general way is the use of some customization method. It should customize the visualization environment for a concrete component model but visualize all applications in a similar way. Favre [18] mentioned the need for such description of a component model prior to the visualization of a component. An example of such initialization method is the use of profiles in UML; more details are in Subsection 4.2.1. In any case, there has to be a way to visualize details bound to a specific component model.

4.2.1 UML 2

The Unified Modeling Language (UML) provides three groups of diagrams to model both static and dynamic features of software [39]. UML is a standard visualization approach that is common through all software companies. It includes a diagram to model the structure of component-based applications called component diagram and defines its visual syntax. UML introduces

[1]http://www.eclipse.org/pde/incubator/dependency-visualization/

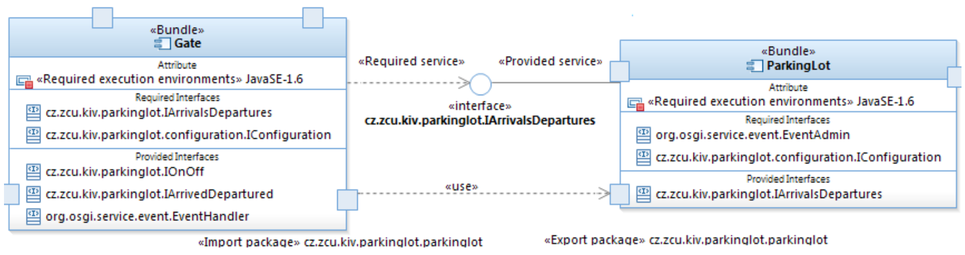

«Bundle»
Gate
Attribute
«Required execution environments» JavaSE-1.6
Required Interfaces
cz.zcu.kiv.parkinglot.IArrivalsDepartures
cz.zcu.kiv.parkinglot.configuration.IConfiguration
Provided Interfaces
cz.zcu.kiv.parkinglot.IOnOff
cz.zcu.kiv.parkinglot.IArrivedDepartured
org.osgi.service.event.EventHandler
«Required service»
«Provided service»
«interface»
cz.zcu.kiv.parkinglot.IArrivalsDepartures
«use»
«Bundle»
ParkingLot
Attribute
«Required execution environments» JavaSE-1.6
Required Interfaces
org.osgi.service.event.EventAdmin
cz.zcu.kiv.parkinglot.configuration.IConfiguration
Provided Interfaces
cz.zcu.kiv.parkinglot.IArrivalsDepartures
«Import package» cz.zcu.kiv.parkinglot.parkinglot
«Export package» cz.zcu.kiv.parkinglot.parkinglot

Figure 4.2: A component diagram with OSGi profile

some level of semantics – it recognizes interfaces, components can have attributes and operations, etc. This semantics is on a meta-model level, thus computer can interpret them correctly and can work with these semantic information. More about the meta-model part of UML will be discussed further in Section 5.2.

As briefly mentioned earlier, UML 2 supports extensions in the form of profiles which can offer a customization of the general component diagram. UML is able to capture enough details about the structure of the application by using these profiles and stay on a general level. This customization is adequate for most of the needs present in component models and has been verified on several component models, for example, CORBA [37] and SaveCCM [45].

An example of a component diagram with OSGi profile is in Figure 4.2. Components are named as “Bundle” instead of general “Component”, ports are used to express imported or exported OSGi packages and interfaces are referred as services. This is a level of customization profiles can offer. One can study book by Eriksson et. al. [17] for more detailed description of UML notation.

The problem is that UML does not meet some of the requirements of component-based development, which would speed up and improve the orientation and understanding of the structure of the component-based application. We summarized these requirements as follows:

- In component-based development, there are roles with very different interests and needs. UML has to use a new diagram for every role in order to provide the exact amount of detail for each of them. Therefore it is time consuming to provide all these roles with appropriate diagrams.

- Stereotypes, which are the power of the UML extension mechanism, are visualized like tags – they only say that the attribute or method belongs to some group. However, all these different attributes/methods are grouped in one place, in one section. But component-based

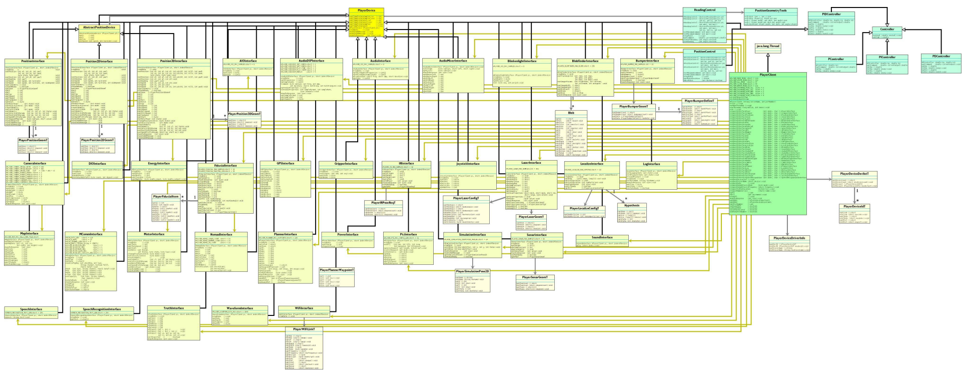


Figure 4.3: An example UML class diagram of java-player taken from http://java-player.sourceforge.net

development, because of its diversity, needs a mechanism to model new types of elements apart from attributes and methods. Moreover, it also needs to show these types separately so one can immediately visually distinguish them.

- UML was designed to be static, to show all information at once and provide the same output on both screen and paper. However, for component assembler and system architect it would be better to have the possibility to switch between level of details. As their work requires them to work with the whole structure and details at the same time.
- The UML diagrams are confusing especially in complex applications. Details of every item are always displayed and every relation is modeled by one line between two items. This can be checked on a class diagram of a relatively simple application in Figure 4.3. A component diagram is comparably confusing as the class diagram depicted in this figure.

Even with these problems, UML is still the best choice for visualization of component-based applications these days. Thus it is not any surprise that there are approaches that extend UML to somehow compensate for these shortcomings, rather than complete alternatives.

Strengths & Weaknesses:

+ The best known and most widely used approach.

+ Can provide sufficient amount of detail.

– All the above-mentioned unsatisfied needs of component-based development.

4.2.2 Layered UML

One of the most problematic feature of UML diagrams is surely the need to have multiple diagrams for the same application or part of an application, which differ only in the number of details provided or in another little way. For example, a simple component diagram without any details to provide better readability of an architecture vs. a component diagram with all details shown to provide enough information to create the whole picture. The solution is to find a way how to accommodate multiple views in one diagram, so the user can easily add (or hide) details or items or change layouts.

In [15], Dimoulin describes such a feature that can extend UML. Dumoulin decided to choose a change-set approach, which he called layered: a final diagram is completed by composing all active change-sets together. These change-sets says what should be changed on the base set. For example: Change set number one can list all the items without any details, change-set two can add details to these items, change-set three can color some items to emphasize them, change-set four can add comments, etc.

This feature is developed as a part of an open source UML tool named Eclipse Papyrus[2]. An example of how these multiple views look is in Figure 4.4.

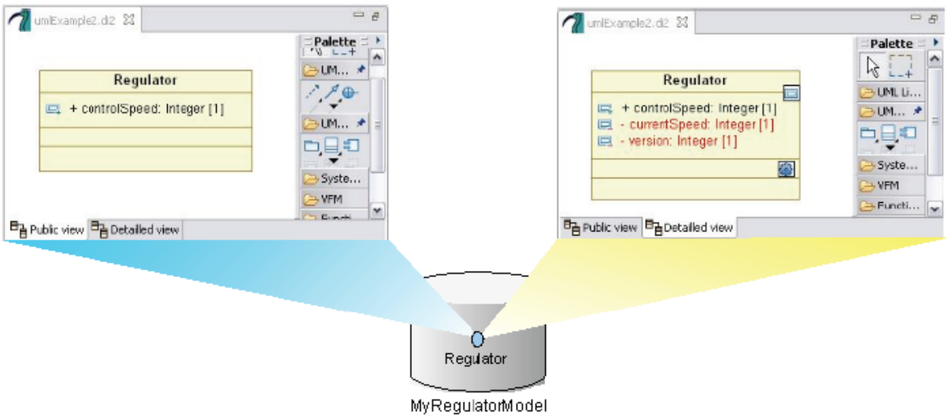


Figure 4.4: Multiple views in one UML diagram [15]

Layered UML diagrams improve usage of UML, because they remove the need for several separate diagrams that have to be maintained, and they enable work with different views seamlessly. But it is still UML and all other problems mentioned in the previous subsection remain valid.

Strengths & Weaknesses:

[2]http://www.eclipse.org/modeling/mdt/papyrus/

+ Can switch between different views interactively.
+ Built on top of UML.
– All the other problems mentioned with UML.

4.2.3 Area of Interest in UML

Area of interest is used to highlight a somehow interesting part of an application. Beylas describes in [9] and [10] how to visualize extra-functional properties inside UML diagrams. In these works he describes why the use of areas of interest is the best approach and he puts special focus on how these areas should be visualized not to disturb the main diagram. You can see a screenshot from the MetricView tool in Figure 4.5. Areas of interest are used to highlight components with the same extra-functional property, for example, the vendors of components. Through this approach it is easier to emphasize shared characteristics between different components, without any unwanted disturbances on the main diagram.

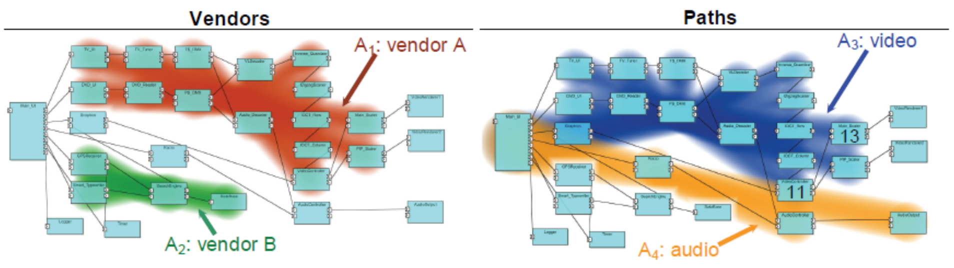


Figure 4.5: Several Areas of Interest in a component diagram [9]

Visualization of metrics is important and the approach described by Byelas can be applied on any graph-based visualization; thus it is even more valuable for our future work. However, it doesn't address any of the problems of UML mentioned above.

Strengths & Weaknesses:

+ Interesting way to visualize extra-functional properties.
– All the problems mentioned with UML.

4.2.4 SoftVision

Telea et al. describe the principles of an interactive visualization framework able to visualize any component-based application in [66]. Later Sillanpaa demonstrate possibilities of this framework named SoftVision in [54]. SoftVision provides the functionality needed to create one's own visualization tool, namely it allows users to: define a new representation of an item (how items are drawn), define new layouts (where items are drawn), pick a call-back to define how visualized data should interact, and write GUI elements that provide operations with items. SoftVision is not only intended for component-based systems; as is apparent from its description in [54]:

> The SoftVision visualization framework is a general-purpose visual environment for browsing and editing graph-based data. Concrete instances of such data are software architectures, component-based systems, network and web structures, and relational databases.

The concrete visualization tool is created in SoftVision by using Tlc script language [41] mostly; Tlc scripts are used to define any setting available in SoftVision. C++ is then used to create new shapes and nodes that are not part of SoftVision. SoftVision adopt the following techniques of interactive visualization: pan, zoom, translate, rotate and fly through. An example of how SoftVision can be used is in Figure 4.6.

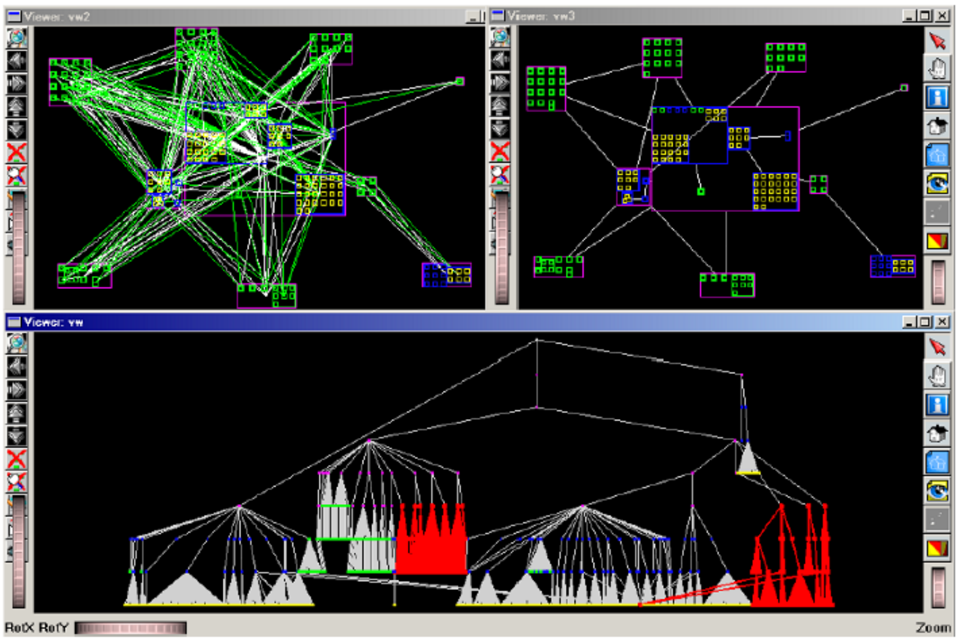


Figure 4.6: SoftVision visualizing architecture on different layouts [54]

The above-mentioned information about SoftVision looks promising, but there is no example of a detailed view anywhere in the article mentioned; instead a number of abstract views are presented. This brings us to believe that SoftVision was supposed to analyze relations in software architectures instead of visualizing the application with needed details. The SoftVision homepage mentioned in the article is offline, and it is not possible to find any other homepage, so information provided in the articles mentioned cannot be verified and we are also unable to test our hypothesis concerning the usage of SoftVision for our purposes. In any case, we believe that SoftVision could not fulfill our goals, because it was too general – it also allowed the visualization of software architectures, networks and web structures. So it would not be able to visualize the structure of component-based applications while providing appropriate details.

Strengths & Weaknesses:

+ Number of means of analysis and visualization.
+ Uses advantages of interactivity.
+ Simple customizability.
– Out-of-date approach without any support.
– Probably could't hold details about components.

4.3 Component Model Specific Visualization

A component model's specific visualization has to introduce its own graphic notation to be able to visualize the specifics of the component model (only a few component models already have one). This results in the need for every developer to learn this notation in order to use it, which complicates the exchange of diagrams between different domain experts.

On the other hand, it has its clear benefits – it can visualize the most important parts, it can show all the details needed and it is able to model all the specifics of the concrete component model. It can be better than any other general visualization approach as it is tailored directly for that one specific component model. It is an interesting example of what is possible when visualization has to support only one component model.

This section will present two different visualizations for two quite distinctive component models – SaveCCM [21] and Palladio [2]. Both these examples are not only a visualization tool, but provide a whole working environment with a several other tools that helps users with the development process.

4.3.1 SaveCCM Visualization

The authors of SaveCCM [21] created a visualization tool that is able to visualize all artifacts of SaveCCM through the development process. This tool is called SaveIDE and is described in [51] and [52]. This tool offers support for the whole development lifecycle, which is, in SaveCCM, composed of design, analysis and synthesis, in this order. The visualization of a structure is present only in the design stage, in which components are designed and connected together to create the architecture of the final application. The architecture editor is part of this tool, and it should offer sufficient work resources. The analysis stage is supposed to test the designed system, by behavior testing, by using timed automata. The behavioral editor is supposed to offer everything for testing of the designed model. In the last stage of synthesis, the application code is generated from the designed and tested architecture.

From our point of view, the architecture editor, which is used for visualization of structure, doesn't offer any advanced interaction technique that could help in the process of learning. The visualization is of course adapted for the needs of SaveCCM with all its visual notations, but otherwise it doesn't offer anything more than a simple representation of a model. Both architecture and behavioral editors are pictured in Figure 4.7.

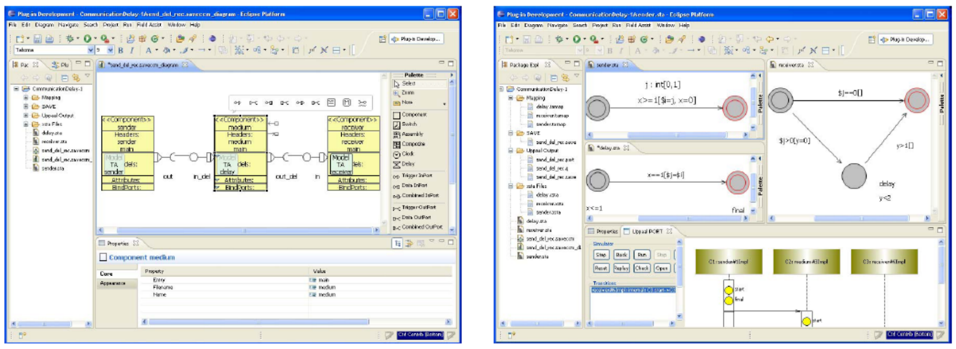


Figure 4.7: SaveIDE – architecture and behavioral editor [51]

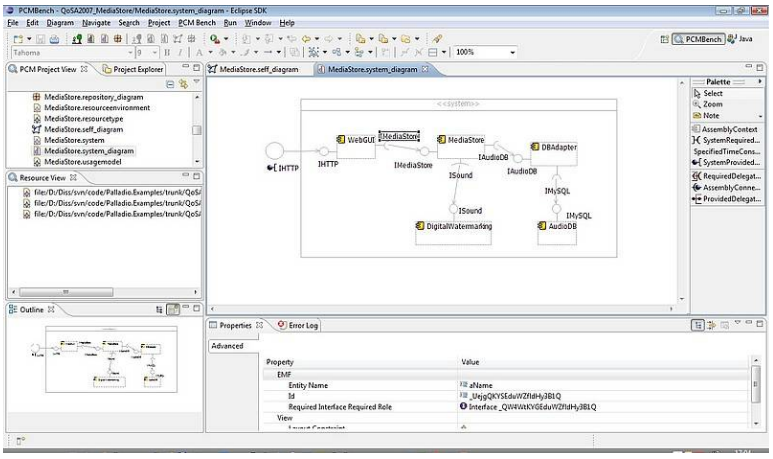
The advantages of SaveIDE as an integrated tool for the whole development process are indisputable, but for visualization of the structure of an application it may not be the best option. This may be the reason why the SaveCCM profile for UML was designed [45]: just to overcome the difficulties of specific visual notation.

4.3.2 Palladio Visualization

The next example, is a pack of visualization tools for the Palladio component model [2] called Palladio Bench. These tools also provide rich capabilities that are adapted for the needs of this component model. The system editor of Palladio applications is shown in Figure 4.8, it doesn't provide much detail, but thanks to other tools that support the development process of

Palladio components[3], it is a valuable supplement.

From other tools, that are also visual, we can name these: repository editor, behavior editor, deployment editor and various comparators. Palladio Bench offers everything that is needed to develop a model and then test and analyze it.



Property	Value
EMF	
Entity Name	aName
Id	_UejgQKYSEduWZfldHy3B1Q
Required Interface Required Role	Interface _QW4WtKYGEduWZfldHy3B1Q
View	

Figure 4.8: Palladio – system editor [51]

The main advantage of Palladio Bench can be again seen more in the other tools, that helps users in development process. Visualization is again only very basic and shows only a minimum information.

4.4 Summary of Current Visualization Approaches

This state of the art overview is based on our extensive research in the field of different visualization tools and approaches. The summary of our findings is that the UML component diagram is number one in visualization of component-based applications and that UML profiles are commonly created to add support for different component models to extend a general component diagram.

There are several approaches that modify properties of plain UML to overcome its shortcomings and we mentioned the most interesting ones. But there is no other visualization approach that could provide more sophisticated visualization than a general UML component diagram, or an even more general "boxes-and-arrows" diagram. However, we found an interesting exception, a SoftVision tool, which was described in subsection 4.2.4.

A general UML component diagram can be visualized in any UML 2 editor:

[3]http://www.palladio-simulator.com/tools/screenshots/

e.g., MagicDraw[4], Papyrus[5], StarUML[6] or IBM Rational Software Modeler[7]. A general "boxes-and-arrows" diagram can be visualized easily thanks to vast number of visualization libraries: e.g., yFiles[8], Protege[9], Neoclipse[10] and Jgraph[11]. These approaches are good when one needs to understand the structure or relations in an application, but for proper understanding and insight they are unusable.

Component model specific visualization approaches can offer sufficient details needed for understanding of the structure of the application, but they all share one obvious and major disadvantage: they are only for one component model. This obstacle doesn't have to be a blocker for some use cases, but a general visualization approach, that could represent details is better and is also the goal of this thesis. However, it is interesting to compare what it is possible to use when one needs to support only one component model.

The best solution, as we see it, is to create approach that can be initialized for a concrete component model and visualize all the applications in a similar way. There is currently only one approach that can do that – UML 2. Due to the imperfections of UML itself, there are efforts to enhance UML and remove these imperfections, but in any case, these efforts doesn't concern components.

[4]http://www.magicdraw.com/
[5]http://www.eclipse.org/modeling/mdt/papyrus/
[6]http://staruml.sourceforge.net/
[7]http://www.ibm.com/developerworks/rational/products/rsm/
[8]http://www.yworks.com/en/products_yfiles_about.html
[9]http://protege.stanford.edu/
[10]http://wiki.neo4j.org/content/Neoclipse_Guide
[11]http://www.jgraph.com/jgraph.html

Chapter 5

Meta-models for the Description of Component-Based Applications

Data that describe components in detail have to be stored in some data structure. It is important that this data structure is able to describe both the surface of a component and the whole component model. The description of the component model is important because it defines what elements are present on component surface and thus helps to keep the description of components on a general level. The need for such description was described by Favre in [18], where he stated that the description of component model is vital.

MOF (Meta Object Facility) [36] provides the required basis for the definition of any such structure. MOF describes four levels of abstraction and itself is positioned on the top abstraction level with the ability to describe other meta-metamodels - i.e. MOF can describe itself. The hierarchy of these abstractions will be described from the bottom up, to illustrate how the level of abstraction rises. This description will be based on Figure 5.1.

The implementation of a concrete component that can be assembled and deployed is the lowest level - M0. When one wants to describe this component, one has to use a prepared structure where the features of this concrete component are represented. Thus an abstraction is created - a model which is at the M1 level. This model is a representation of a component-based application. The structure that describes what types of features can be added to component and what is permitted in the model is a meta-model, which is on M2 level. A meta-model is a representation of the component

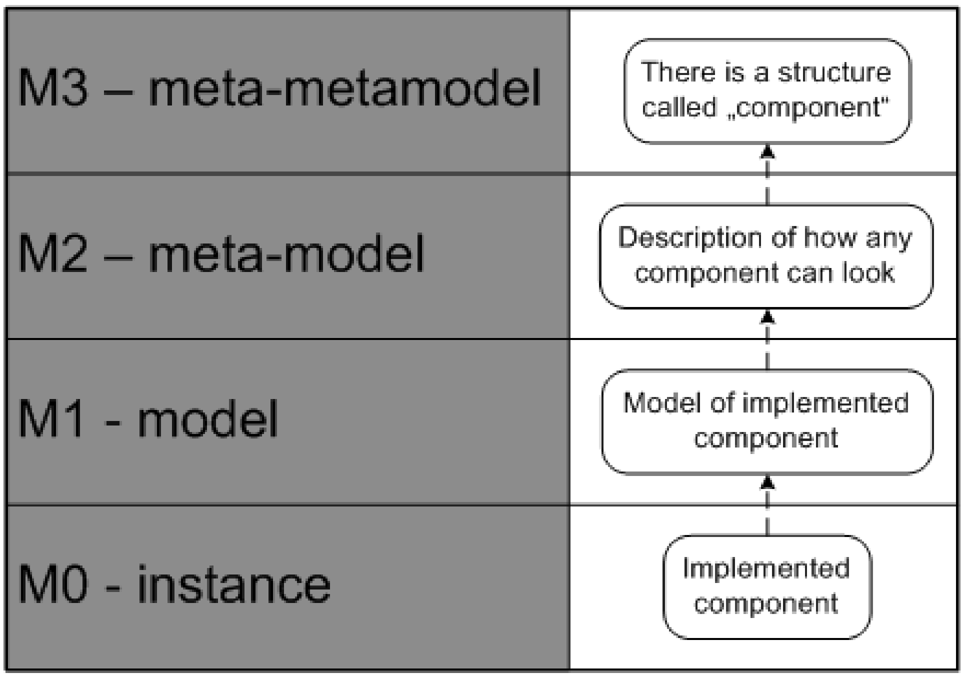

M3 – meta-metamodel
There is a structure called „component"
M2 – meta-model
Description of how any component can look
M1 - model
Model of implemented component
M0 - instance
Implemented component

Figure 5.1: MOF abstraction levels mapped on the component domain

model. Finally on top of all of this is M3 level which defines that there is a structure called “component” that will be used for the representation of real world components, and that there is e.g. a structure called “element” that will be used to define different types of features that can be present on component’s surface.

This MOF-based introduction can be generalized by stating that for the description of component-based application it is needed to create a meta-metamodel to describe both the component model and component-based applications. Another approach could be to use a meta-model that supports extension mechanism to extend a general description by more details. This was already mentioned as an initialization of concrete component model in Section 4.2.

In the following sections we will present research models, as an overview of related works; the UML meta-model, as a reference state of the art; and the ENT meta-model, as the solution which backs our component visualization approach (Chapter 6). The ENT meta-model will be described thoroughly as it was extended as part of this work, the UML meta-model will be shortly described as it is a well known standard and other research models will be only briefly discussed.

5.1 Research Models

In [48] Rastofer describes a meta-model capable of modeling various component models to unify their basic features. This work analyzes the shared features of different component models, to find a means how to describe them in a minimalistic way. The meta-model is component-based and is able to describe itself, thus it terminates the meta-level hierarchy. The main benefits of this approach for us lay in the minimalistic representation of any component model through three types of constructs - component, port and connector. The component model is then described by characterizing what the components of this component model can do and how they can communicate. Rastofer also offers a simple visual notation of this meta-model to make modeling of different component models easier and provides several example models.

Crnkovic describes in [13] an advanced framework able to classify any component model from various angles. This work is highly analytical and deals with a number of different component models together with their characteristic features compared all together. The framework described in this work identifies four major categories in which component models behave differently - lifecycle, constructs, extra-functional properties, domain - and identifies the individual elements of these categories. This framework offers a complex survey of how component models can vary, thus it is an ideal basis for future analysis of shared features needed to design a meta-model able to describe them.

5.2 UML 2 Meta-Model

UML meta-model is defined in [38]. It is closely bound to MOF [36] because it is not only defined by MOF, but also it is a part of core specification. This relationship is shown in Figure 5.2 where one can clearly see that UML, MOF and Profiles use the same shared common core. However UML is still an instance of MOF from architectural point of view – therefore they are on different meta levels.

The Core package contains the packages PrimitiveTypes, Abstractions, Basic, and Constructs. The *PrimitiveTypes* package contains a number of predefined types that are commonly used in meta-modeling. The *Abstractions* and *Constructs* packages contains a number of fine-grained packages with only a few meta-classes each. However, *Abstractions* provide mostly abstract meta-classes that are highly reusable, while *Constructs* provide mostly concrete meta-classes rather than abstract ones. Finally, the *Basic* package contains a subset of *Construct* that is used primarily for XMI purposes.

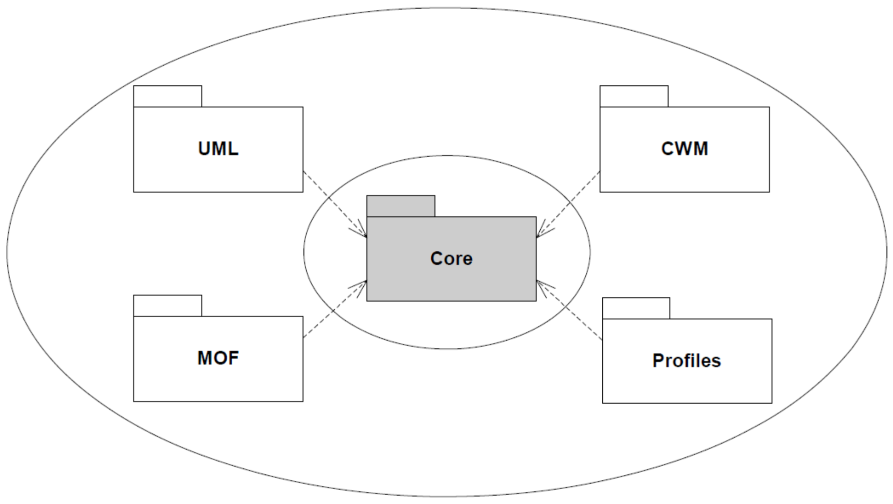


Figure 5.2: The role of common core [38]

UML meta-model define keywords like *package*, *class*, *relationships* which are further extended to define more concrete keywords like *component*, *interface* or *dependency*. This is the way how more concrete classes are created – the basic metaclasses are extended. UML meta-model is extremely complex and should be studied thoroughly to gain understanding however, this is out of the scope of this thesis.

5.2.1 Components

The UML meta-model defines the “component” classifier from UML version 2.0 and it offers a basic features shared across all component models. It is important to keep in mind that all information about components are kept by the UML model in order to use them, for instance to visualize them in a UML visualization tool. The basic meta-model for component diagram is shown in Figure 5.3.

One can see that this structure is really very general and does not offer much more than the possibility to define provided interfaces, required interfaces, dependencies and model basic relationships. Such structure can be used to model any component-based application, however it will not be able to show any details. Please note that *Component* extends *Class*, therefore it inherited all its options – component can define attributes, methods, etc.

For better results, one has to use *Profiles* in order to be able to model component model specific features.

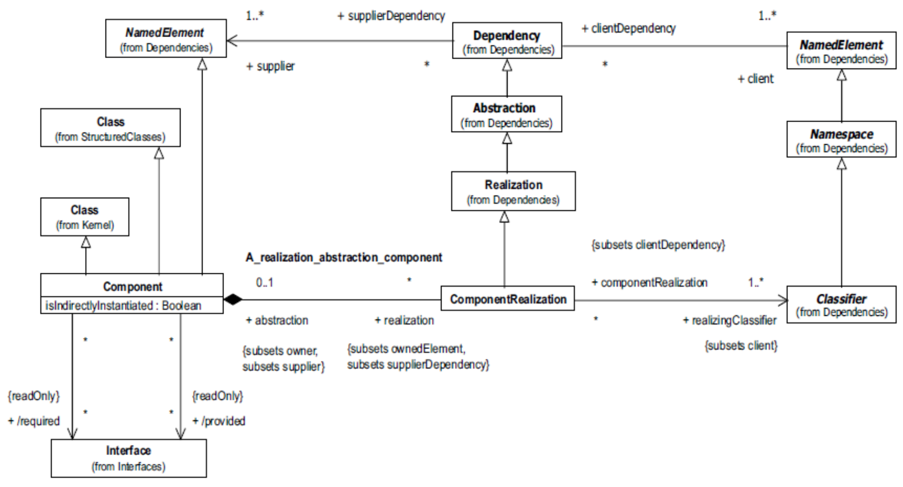

NamedElement (from Dependencies)
1..*
+ supplierDependency
+ supplier
*
Dependency (from Dependencies)
+ clientDependency
*
1..*
NamedElement (from Dependencies)
+ client
Class (from StructuredClasses)
Abstraction (from Dependencies)
Namespace (from Dependencies)
Realization (from Dependencies)
Class (from Kernel)
{subsets clientDependency}
A_realization_abstraction_component
0..1
*
+ componentRealization
1..*
Component
isIndirectlyInstantiated : Boolean
ComponentRealization
Classifier (from Dependencies)
+ abstraction
+ realization
*
+ realizingClassifier
{subsets owner, subsets supplier}
{subsets ownedElement, subsets supplierDependency}
{subsets client}
{readOnly}
+ /required
{readOnly}
+ /provided
Interface (from Interfaces)

Figure 5.3: Classes defined for Component diagram [39]

5.2.2 Profiles

We already mentioned that profiles can be used to extend the descriptive capabilities of the UML – to initialize component diagram to support some specifics of a component model. On this meta-model level we can be more precise – profiles provide mechanisms to extend the existing meta-classes to adapt them for different purposes. This includes mentioned initialization or in other words to customize the UML for various different platforms or domains. Profiles extend the common core of UML/MOF and can therefore use already defined meta-classes. The structure of *Profiles* package is shown in Figure 5.4.

It contains all classes needed to define a new profile. The most important classes, from our point of view, are *Profile* and *Stereotype*. *Profile* basically only defines a set of *Stereotypes*. *Stereotype* is the description of new elements that are used for the tailoring for concrete platform/component-model. *Stereotype* extends *Class*, therefore it can use features defined for classes – attributes, relationships, constraints, etc.

UML Profiles offer three types of extension mechanisms that are used in new profile definition process. We name them repeatedly from a modeling point of view, how they are mentioned in various UML handbooks, for example Eriksson [17]:

1. **Stereotypes** allow designers to enrich the vocabulary of UML by extending an existing element. These stereotypes can be applied on represented items, to mark their special characteristics. Stereotypes can have their own properties and settings, that are inherited from

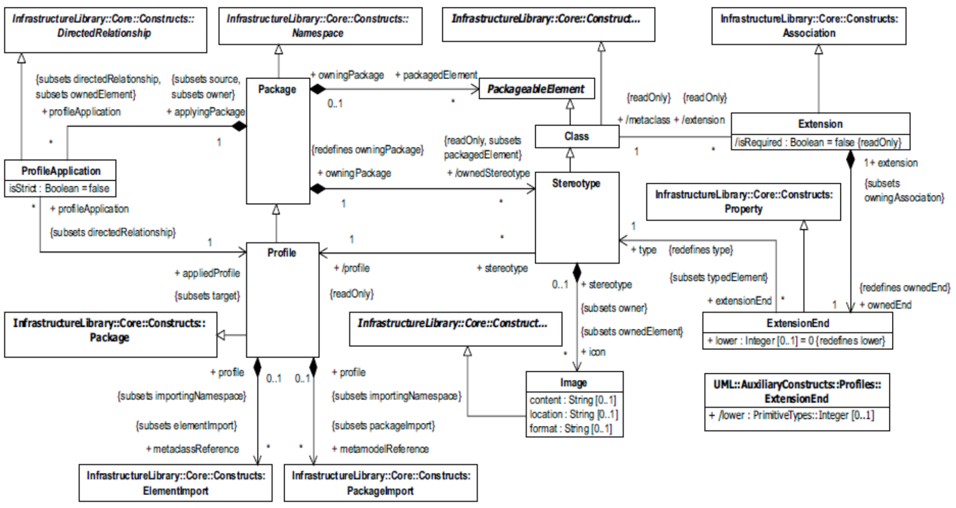


Figure 5.4: Classes defined in Profile [38]

stereotype to item.

2. **Tag definitions** are properties of stereotypes. The values of tag definitions are referred to as tagged values.

3. **Constraints** define the conditions or restrictions to which a model element has to conform.

These extension mechanisms may be sufficient to represent specific details of any component model, but for future analytical work required from interactive tools they are inadequate. Such requirements could be met only by modifying the UML meta-model directly. This approach was described by Perez-Martinez in [44]. The author used this "heavyweight" extension of UML to provide a better description of C3 architectural style [53].

The modified UML meta-model could offer everything needed by advanced interactive visualization approach but it would require a lot of changes. It would also become a constant maintenance problem because UML is a mature and complex meta-model and every modification brings new and unknown dangers. This is why we believe that it is better to design a new meta-model, rather than to modify an existing one, even if such an existing model has undisputed qualities.

5.3 The ENT Meta-Model

The ENT meta-model is a MOF M3 model defining the structures of component models and component-based applications. Its version extended by this thesis is described in [3]. Originally the model supported only description of single components, without the respect to relations - inter-component bindings and hierarchical composition of components. Extensions made by author of this report are described in subsections 5.3.4 and 5.3.4.

ENT's main characteristic is the use of the faceted classification approach [47] to represent components in a way which is flexible enough for users with different interest. A key structure used in the meta-model is the ENT *classifier*, which is a tuple of identifiers which characterize any component interface element from several orthogonal aspects related to user perception.

The ENT meta-model is structured into two levels: on the *component model level* the main characteristic features of a given component model are defined, on the *application level* the concrete components, their interface elements and their bindings in an application are captured.

The whole ENT meta-model formal description, which is described in the following subsections, was analyzed in order to create a corresponding MOF model. This model was implemented using EMF (Eclipse Modeling Framework) [62]. The process of this implementation is described in [56]

5.3.1 Overview of the Meta-Model

Let us start with a brief overview of the meta-model in plain English; the following subsections will then provide the exact definitions. The structural hierarchy of the meta-model starts with a *component model* as a set of component types. A *component type* is defined by a complete minimal set of *definitions of traits* which describe the possible kinds of interface elements which the component type can support. The traits declare the language meta-type and ENT classifier of these elements, capturing their commonalities like the users do.

As an example, there is only one component type in OSGi called "bundle", with traits $\{export\_packages, import\_packages, provided\_services, required\_services, etc.\}$. The ENT meta-model enforces this structuring of component interface (as opposed to a flat collection of items, cf. Figure 6.4) because it is quite natural for developers to think of e.g. all component's provided services as a group, regardless of their concrete interface types and location in the specification source. In Enterprise JavaBeans on the other hand several different component types can be identified – SessionBeans, MessageDriven-Beans or Entities. The component types, as well as trait's characteristic meta-type and classifier, are therefore based on a human analysis of the concrete component model and its component specification language(s).

At the level of a concrete application, a *component* implementation then conforms to one of the component types defined by its component model.

Each component has a set of concrete *interface elements* manifest on the visible surface of its black box. These elements populate some or all of its actual *traits*, which again conform to the corresponding trait definitions. The component also holds the *connections* of its elements to the counterpart elements in client and/or supplier components, and – in case of hierarchical component models – may list the *sub-components* it is composed from.

In many component models, several run-time *instances* of a concrete component can be created, each with unique identity. The ENT meta-model does not deal with component instances because its domain is the level of component models and component application design, rather than the run-time instantiation level.

The rest of this section provides a formal definition of these structures, in a top-down fashion.

5.3.2 Classification System

The ENT meta-model uses a faceted classification system for characterizing various aspects of component interface elements, with eight facets called “dimensions”. These dimensions have predefined values and each dimension represents a different point of view on a component.

Definition The **ENT classification system** is a collection of facets $Dimensions_{ENT} = \{dim_i, i = 1..8\}$ where the dim_i are:

- Nature = {syntax, semantics, extra-functional}
- Kind = {operational, data}
- Role = {provided, required, neutral, provided_and_required}
- Granularity = {item, structure, compound}
- Construct = {constant, instance, type}
- Presence = {mandatory, permanent, optional}
- Arity = {single, multiple}
- Lifecycle = {development, assembly, deployment, setup, runtime}

The **ENT classifier** is a tuple $K = (k_1, k_2, ..., k_D)$ where $k_i \subseteq dim_i, dim_i \in Dimensions_{ENT}, D = \mid Dimensions_{ENT} \mid$.

This classification system and the classifier structure are used in the trait and category set definitions, presented in the subsequent paragraphs.

5.3.3 The Component Model Level

Identification of different component models and the types of components they define forms the top level of the meta-model.

Definition A **component model** is the pair $M = (\mathit{name}, C_S)$ where $\mathit{name} \in \mathit{Identifiers}$ is the model's name and $C_S = \{C_{i,def}\}$ is a set of component type definitions.

Component types consist mainly of trait definitions that declare the kinds of elements (features) the concrete components can have on their surface. Traits thus helps to fully characterize components of such type. For example, OSGi components (cf. Section 5.3.3) have traits *Export packages*, *Provided services*, *Import packages*, etc.

Definition A **component type** is a tuple $C^{def} = (\mathit{name}, \mathit{tagset}, T)$ where $\mathit{name} \in \mathit{Identifiers}$ is the name of the component type, $\mathit{tagset} = \{\mathit{tag}_i\}$ is a finite set of extra type information items ("tags"), and the $T = \{T_i^{def}\}$ where i is a finite index is the set of the component type's trait definitions (also called "trait set").

The tags in the tagset are quaternion $\mathit{tag}_i = (\mathit{name}_i, \mathit{metatype}, \mathit{valset}_i, d_i)$ where $\mathit{name}_i \in \mathit{Identifiers}$, $\mathit{metatype} \in \mathit{Identifiers}$ is the meta-type of the tag, valset_i is the set of its possible values, and $d_i \in \mathit{valset}_i \cup \{\epsilon\}$ is the default value (ϵ means "no default"). Tags capture pieces of information that are important for the component model and cannot be described using traits, e.g. component's persistence and transactionality as used in Enterprise JavaBeans.

The component types of one component model must be distinct: $\forall C_i, C_j \in M.C_S, i \neq j : C_i \neq C_j \implies C_i.\mathit{name} \neq C_j.\mathit{name}.$

Definition A **trait definition** is a tuple $T^{def} = (\mathit{name}, \mathit{metatype}, K, \mathit{tagset}, \mathit{extent})$ where $\mathit{name} \in \mathit{Identifiers}$ is the trait's name, $\mathit{metatype} \in \mathit{Identifiers}$ is the meta-type of the component interface elements grouped by this trait, K is their ENT classifier, $\mathit{tagset} = \{\mathit{tag}_i\}$ is the finite set of allowed tags of these elements, and $\mathit{extent} \in \{\mathit{one}, \mathit{many}\}$ defines the maximum number of elements in the trait[1].

Consistency rule: Traits of one component type must be distinguishable by name, i.e. $\forall T_i^{def}, T_j^{def} \in C^{def}.T, i \neq j : T_i^{def}.\mathit{name} \neq T_j^{def}.\mathit{name}.$

[1]For simplicity, we do not use concrete numbers, ranges and similar features in extent specification.

The *metatype* of the trait's elements (such as "interface" or "event") may be related to or derived from the name of the corresponding non-terminal symbol in the grammar of the component's interface specification language particular for the trait. The *tagset* has the same definition and meaning as that of the component, described above, except that the concrete tag values are meant to be assigned to individual elements (not to the trait).

The ENT classifier K describes the classification properties of the trait's elements – this is a unique aspect and key concept of the ENT meta-model, capturing the human-perceived similarity of the elements grouped by a trait.

Concerning the consistency rule, it is actually preferred that traits are distinguished by their classifiers only, i.e. the following stronger assertion holds: $\forall T_i^{def}, T_j^{def} \in C^{def}.T, i \neq j : T_i^{def} \neq T_j^{def} \implies T_i^{def}.name \neq T_j^{def}.name$. There may however be cases when the ENT classification scheme does not provide enough characteristics to reliably distinguish traits. Then, distinguishing by names is the only practical option and this is reflected in the definition.

When the component model level description is designed according to the ENT meta-model, a set of data structures for modeling component-based applications is prepared. These data structures can fully describe all components implemented in the given component model and have to be created manually after analysis of modeled component model. The following section illustrates the ENT component model definition for the OSGi framework.

Example: The OSGi Component Model and Application

To illustrate the ENT structures, this section presents a subset of the representation of the OSGi component model [40] plus examples of behavioural and extra-funcional element traits. OSGi was chosen for its industrial relevance, simplicity and ubiquity.

Component Types OSGi has only one component type called **Bundle**. Bundle can have two additional tags originated in manifest file.

1. **Bundle**
   - **tagset**: symbolic_name, version
   - **T**: { export_packages, import_packages, provided_services, required_services, native_code, require_bundles, required_execution_environment, use_packages}

Trait Definitions For demonstration purposes we provide the definitions of just four traits here, see [67] for a complete analysis of OSGi ENT representation:

1. **export_packages**
   - **metatype**: package
   - **K**: ({syntax}, {operational}, {provided}, {structure}, {type}, {permanent}, {multiple}, Lifecycle)
   - **tagset**: version, parameters
   - **extent**: many
2. **import_packages**
   - **metatype**: package
   - **K**: ({syntax}, {operational}, {required}, {structure}, {type},{permanent}, {single}, Lifecycle)
   - **tagset**: bundle_symbolic_name, bundle_version, kind, version_range
   - **extent**: many
3. **provided_services**
   - **metatype**: interface
   - **K**: ({syntax}, {operational}, {provided}, {item}, {instance},{optional}, {single}, Lifecycle)
   - **tagset**: service_filter
   - **extent**: many
4. **required_services**
   - **metatype**: interface
   - **K**: ({syntax}, {operational}, {required}, {item}, {instance},{optional}, {multiple}, Lifecycle)
   - **tagset**: service_filter, service_arity
   - **extent**: many

Behaviour and Extra-Functional Properties Traits can also represent other than functional elements, for example a quality of service aspect (e.g. [29]) or the expected call sequence protocol [46]. These traits must have value *semantics* respectively *extra-functional* in the dimension *Nature* of the ENT Classification. Sample trait definition for such elements are provided below:

1. **response**
   - **metatype**: attribute
   - **K**: ({extra-functional}, {data}, {provided}, {item}, {constant}, {mandatory}, {single}, {runtime})
   - **tagset**: $\emptyset$
   - **extent**: many
2. **protocol**
   - **metatype**: regular-expression
   - **K**: ({extra-functional}, {operational}, {provided}, {structure}, {type}, {optional}, {single}, {assembly, runtime})
   - **tagset**: $\emptyset$
   - **extent**: one

Example OSGi Application In the subsequent sections we will refer to (parts of) a simple example OSGi application called Parking Lot. It consists of four components as illustrated in Figure 5.5, the architecture should be self-descriptive.

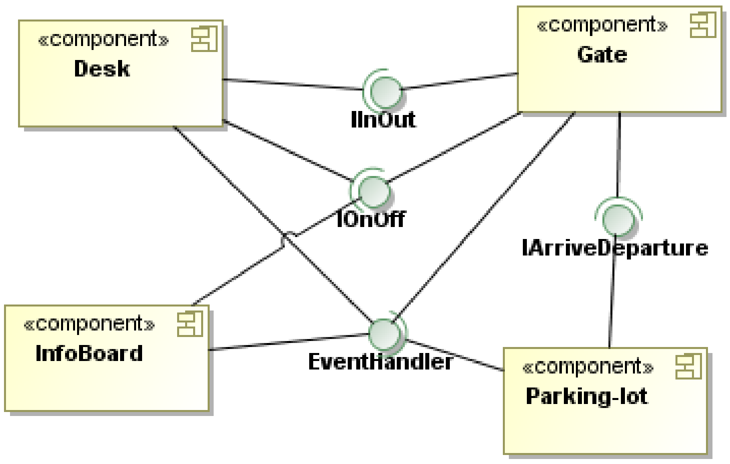


Figure 5.5: Component application example — Parking Lot (OSGi application)

5.3.4 Application Level

This level of the ENT meta-model provides modeling constructs for concrete components and applications built from them. The component model level has to be already defined because the application level references its elements. These references assign meaning to the application elements; in particular, the set of traits of a concrete component is gained by assigning it the corresponding component type.

Definition A **component application** is a direct acyclic graph $A = (C, B, m)$ where $C = \{c_i, i \in \mathbb{N}\}$ are components, $B = \{b_i, i \in \mathbb{N}\}$ their bindings, and $m \in C$ is a main component. We use the term **application context** for a set of all components $A^* = \{c_i, i \in \mathbb{N}\}, A.C \subseteq A^*$ existing in the environment where the component application is deployed.

A *consistent (resolved) application* is such that has all non-optional required elements bound to provided ones within the given context and all its components' inheritance parents exist in the context.

We do not model additional pieces of information associated with applications, like configuration properties, access control lists, and similar – these are used at run-time which is out of scope for ENT meta-model.

Individual Components

In this section an example of the Gate bundle (see Figure 5.5) will help to ilustrate the representation of component information in the ENT meta-model structure. The manifest file of this bundle is present in Figure 5.6.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Gate
Bundle-SymbolicName: Gate
Bundle-Version: 1.0.0
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Require-Bundle: Parkinglot;version="1.0.0"
Import-Package: cz.zcu.kiv.parkinglot.parkinglot;version="1.3.0",
 org.osgi.service.event;version="1.2.0"
Export-Package: cz.zcu.kiv.parkinglot.gate
```

Figure 5.6: Manifest file for Gate bundle

Definition A concrete **component** is a tuple $c = (name, C^{def}, G, T, P, S)$ where $name$ is the component's name, C^{def} is the (reference to) the appropriate component type, $G = \{(name_i, value_i)\}$ is the set of its tags, $T = \{t_i\}$ is the concrete trait set of the component with traits as defined below, P is a finite, possibly empty set of (references to) concrete components which are c's inheritance parents, and S is a finite, possibly empty set of c's subcomponents and their delegation bindings (see subsection 5.3.4 below).

The following consistency rules must hold:

- $\forall (n_i, v_i) \in c.G\ \exists tag_j \in C^{def}.tagset : n_i = tag_j.name \wedge v_i \in tag_j.valset$, i.e. tags are taken from component's type tagset;
- $\forall p \in P : p.C^{def} = c.C^{def}$, i.e. the parents are of the same component type.

It is also natural that both c and all its sub-components belong to the same component model.

By component interface **element set** $E(c)$ we will understand the set of all interface elements (as defined below) contained in the specification of concrete component c. In case of component inheritance, it is the union of element sets of the transitive closure of c and all its inheritance parents. Subsets $E^P(c)$ and $E^R(c)$ of the element set denote the *provided* and *required elements* of c where it holds that $E^P(c) \cap E^R(c) = \emptyset \wedge E^P(c) \cup E^R(c) = E(c)$.

This representation is a complete model of a concrete component, by which we mean that the original specification of the component can be fully reconstructed from the representation.

Concrete component's trait is a named set of its interface elements with the same meaning, as given by their meta-type and ENT classifier.

Definition A component interface **trait** (of a concrete component c) is a pair $t = (T^{def}, E)$ where T^{def} is a (reference to) the trait definition and $E \subseteq E(c)$ is a subset of component's interface elements.

Consistency rules: It must hold for a given component c that

- $E(c) = \bigcup_i t_i.E, t_i \in c.T$ and $\forall t_i, t_j \in c.T, t_i \neq t_j : t_i.E \cap t_j.E = \emptyset$, i.e. that the traits together contain all its elements without duplicates
- $\forall t \in c.T : t.T^{def} \in c.C^{def}.T$, i.e. traits are defined by its component type.

Traits group the interface elements of a component even if in the source these may be specified in various places – either within one specification file (e.g. a SOFA ADL, disregarding the particular ordering of declarations), or even in several ones (e.g. OSGi manifest plus declarative services' `component.xml`).

Traits alone do not say anything about the features of the particular component – they have only grouping purpose and through the reference to their trait definitions give meaning to all interface elements contained in it.

Definition An **interface element** e of a concrete component c with specification written in language L is a tuple $e = (name, type, G)$ where $name \in$

T^{def} = imported_packages,
$E = \{cz.zcu.kiv.parkinglot.parkinglot, org.osgi.service.event\}$

Figure 5.7: The *imported_packages* trait of the *Gate* bundle in ENT representation

$\mathit{Identifiers} \cup \{\epsilon\}$ is the (possibly empty) element's name, $\mathit{type} \in L$ is a language phrase denoting its type, and $G = \{(n, v)\} \subset \mathit{Identifiers} \times \mathit{Identifiers}$ is the (possibly empty) set of element's concrete tags.

Consistency rule: $\forall e \in t.E, \forall g \in e.G\ \exists d \in t.T^{def}.tagset : g.n = d.name \wedge g.v \in d.valset$, i.e. the tag values of elements in trait t must be taken from the value set in the trait definition.

A specification element is a complete representation of one component interface feature identified by language *name* and/or *type*. All its parts are directly related to its specification source code (the human classification and understanding of an element is attached to its containing trait). Operations on them are therefore subject to the syntax and typing rules of the language L used for the component interface specification.

The tags represent additional semantic or other extra-functional information pertaining to the particular element (not to its type), like the `readonly` or `final static` keywords. They are important if one needs to e.g. precisely compare two elements or re-generate a valid source code for the element. Note that the element's tags are defined in its trait definition, since all elements of one trait necessarily have the same set of tags.

name = cz.zcu.kiv.parkinglot.parkinglot,
type = package,
$G = \{(version, 1.3.0)\}$

Figure 5.8: The *parkinglog* element of the *imported_packages* trait in ENT representation

Component Bindings

To model bindings between components within the application, we use a set of connections which keep information about source element, target element and which direction information flows (provided / required).

Definition Let us have a consistent component application A. The **application connection set** is a finite set $B = \{b_i, b \in \mathbb{N}\}$ where $b = (e^s, e^t)$:

$\exists c_i, c_j \in A.C : e^s \in E^R(c_i), e^t \in E^P(c_j)$ i.e. the connections (arcs in the application graph) lead from required to provided elements.

The **connection set of a component** c is a set of connections which have incidence with the component: $B(c) \subseteq B$, $\forall b \in B(c)$ either $b.e^s \in E^R(c)$ or $b.e^t \in E^P(c)$.

The connection set of a component makes it possible for every component to be aware of all connections realized by its elements, both provided and required.

e^s = Gate::exported_packages::cz.zcu.kiv.parkinglot.gate,
e^t = Desk::imported_packages::cz.zcu.kiv.parkinglot.gate

Figure 5.9: The service *cz.zcu.kiv.parkinglot.gate* bound to bundle *Desk* in ENT representation

Hierarchical Components

Some component models such as SOFA [8] use hierarchical decomposition which means that composite components can be recursively composed from other components. Components which are not composed from any other components are called primitive components.

For composite components, a special set of connections needs to be modeled: the subsumption and delegation bindings between the composite component interface elements and its sub-components.

Definition For a given component c in application A, the pair $S = (S^c, S^d)$ in component's tuple captures the **inner architecture** of its composition. $S^c \subset A.C, c \notin S^C$ is the set of sub-components. The $S^d \subseteq B$ is a set of delegate/subsume binding pairs, $S^d = \{(e^c, e^s) \mid e^c \in E(c),\ e^s \in E(s) \cdot s \in S^c\}$, i.e. the e^c and e^s elements belong to the composite component and one of its sub-components, respectively.

Consistency rule (added to those in Definition 5.3.4): $\forall (e^c, e^s) \in S^d : e^c \in c.t_m, e^s \in s.t_n, t_m.T^{def} = t_n.T^{def}$, i.e. elements in subsume/delegate pairs belong to traits with the same trait definition.

For example, suppose that the *Parking-lot* component from Figure 5.5 was in fact hierarchical. The handling of client's requests on the `IArriveDeparture` element could be delegated to an equally-typed element in a *Arrivals* sub-component. This would be expressed as an inner architectural binding *(Parking-lot::IArriveDeparture, Arrivals::IArriveDeparture)*. Both elements would belong to the "provided-services" trait of their components.

5.3.5 Structuring Level: Category sets

Some traits and elements could be at particular times considered as unnecessary information when analyzing a model of component-based application. For example, software architects are interested in other information than programmers. By using all information contained in both layers of an ENT-based model there could also be a danger of confusion when representing big and complex applications.

After representing a component-based application according to the Application level, the ENT classifier allows us to organize the model information using so called *category sets*. These sets are defined by selector operators on the trait classification which say how to group and filter traits.

Definition The **category set** over an ENT model is a pair $Catset = (name, \{(c, K, f)\})$ where $name, c \in Identifiers$ are the names of the category set and its categories, and $f = K \times T^{def} \rightarrow boolean$ is a function which determines whether the given trait definition fits the (partial) classifier K.

For example, the E-N-T category set defined in Figure 5.10 has three groups. In the first group are elements that are contained in traits with $role = \{provided\}$ in their classifier (this means those elements which the component *exports*). Required elements are similarly grouped as *needs* and elements that are both provided and required are called *ties*. This category set gave the name to the ENT meta-model, as it captures the most fundamental split of any component's interface element set.

Figure 5.10: The ENT category set

E-N-T (Exports-Needs-Ties)
$E : K = \{(role = \{provided\})\}, f = matches$
$N : K = \{(role = \{required\})\}, f = matches$
$T : K = \{(role = \{provided, required\})\}, f = matches$

More category sets are presented in [3], and category sets can be created by any user of the ENT meta-model if another point of view is needed.

Chapter 6

Advanced Interactive Visualization Approach

In this chapter a new visualization approach (Advanced Interactive Visualization Approach – AIVA) will be proposed. AIVA provides an alternative to the UML component diagrams – it is able to visualize the structure of any component-based application. It is driven by the aims defined in the Introduction Section 1.2 which were all addressed. These aims were:

Visualize the structure of any component-based application as a graph diagram – because it is common practice in visualization of structure thus users are used to it. AIVA honors this aim.

Visualize a sufficient amount of detail – because details are needed to get the whole picture. AIVA uses the ENT meta-model as a data layer. The ENT is able to describe any component and any structure in much detail. A visualization using it can benefit from its advantages, which were mentioned earlier. Moreover, the ENT meta-model provides well structured and categorized information.

Provide ways to filter these details and work on different levels of detail interactively – because otherwise it is necessary to create multiple diagrams for the same structure, differing only in the number of details. These different diagrams or more precisely views should rather be provided "on the fly" driven by the needs of the user. The visualization should not be forced to always show all the information, but modify the provided information as the requirements changes. AIVA uses the ENT meta-model and interactive techniques to address this aim.

Maximize the advantages of interaction to boost the learning process – because it should help to understand the application faster and more easily. It should be able to help in any situation where it is important to keep the scope of the application under control, while having at the same time, access to the details. These requirements are met in AIVA thanks to focus on human-computer interaction and by maximizing the advantages of several interactive techniques. Such visualization is not supposed to be used on the paper, however it can offer much better experience on computer.

The content of this chapter covers research challenges, that drove the development of this approach; brief description of the approach itself, to emphasize how to solve the challenge; and lastly a thorough description of concrete solution.

6.1 Research Challenge

This section will summarize and discuss more deeply research challenges that were solved as a part of this thesis. These are the problems that we had to face before we even formulated the aims. The Introduction just shortly outlined them and some were already discussed in more depth. These research challenges were:

1. Diagrams are more complex then other known software structures.
2. Component details make diagrams even more complex than class diagrams are. It is thus because components may have much more types of elements.
3. Component-based software engineering is very complicated - there are dozens of component models and it is very hard to visualize them in a general way.
4. Developer roles have very different requirements on information content.

We already discussed challenges with different developer roles sufficiently in Section 4.1. The research challenge in that section is in how to provide different types of information for these different development roles. However other challenges are still just vaguely defined – thus they will be defined in following subsections.

6.1.1 Complexity of Diagrams

We see one of the biggest research challenge in more complex diagram structure that needs to be visualized – compared to class diagrams, for example. To explain this problem more in depth, let's discuss class diagrams more as it gives us a very good contrast. Class diagrams recognize four types of relationships: associations, generalizations, dependencies and abstractions/realizations. One class is connected to the second class commonly only with one line that clearly defines their relationship. There may be exceptions, however, they are not very common. This results in quite clear structure of nodes with reasonable density of connection lines.

On the other hand components are more complex, they can be built from dozens of classes and provide a variety of different features. Components may provide the whole packages of classes, interfaces as contracts for communication, event queues, semantics, extra-functional properties or something so unique like behavior protocol [46]. And other components may require a variety of them for their functionality. This means that relationships between two components are not so straightforward as between two classes. There are several levels of possible contracts between them, which should be visualized. One component can require from second component four packages, three interfaces and an event queue – eight connection lines that will represent this complex relationship. This results in a diagram with much higher density of connection lines – when considering the diagram with the same number of nodes.

It is common that applications have some core components, as a result of class encapsulation. These core components contain, as the name suggest, all the core functionality. Such core components are usually connected with a large number of other components, that require their core functionality. These components might offer more than one package or interface. For example, component "OSGi Services" provide dozen packages from which are three commonly required by other components. This really makes a diagram very hard to read.

A research challenge is in how to visualize these complex component diagrams. They are obviously more complex than standard class diagrams. Thus more advanced techniques should be used in order of keep them readable. Such technique should lower the complexity without losing any information.

6.1.2 Complexity of Components

Components should be compared to classes again to show that their demands are higher and so is the complexity. Classes are basically described by their attributes and methods which are two groups of different elements. Components in its simplest essence may have some elements that they provide and require and some to characterize itself. However provided interface is not the same thing as a provided event topic – thus it should be differentiated more deeply. Therefore components might be much more complex than classes, depends on the component model used. For example, OSGi has eight groups, which we identified (see Appendix B).

A research challenge is in how to visualize components. Three groups of elements will not express the character of component simple enough. However, ten groups might be confusing and cause a lot of visual clutter. There should be a solution that would provide enough details about these different types of groups while remains easy to read.

6.1.3 Problems with CBSE

Problem with CBSE is that every component model can define its own components that might behave completely differently and have quite distinctive features. Chapter 2 introduced the basic problems – that components and component models are rather different. Later Chapter 4 highlighted problems with unified visualization, and lastly Chapter 5 followed with reasoning why it is hard to only desribe components in a general way.

A research challenge is in visualization of components structure in diagram with details. It is really hard to find a general description that would suit the needs of all component models while providing necessary details.

6.2 Conceptual Structure

The AIVA interactive visualization is built on several principles that help us to achieve mentioned goals. These principles are described and analyzed by Holt in [24] and Meyer in [34]. Firstly several interactive techniques were studied to maximize the impact of interactivity and computer use. An overview of a lot of these techniques is provided in [71]. As we already mentioned in the introduction of this chapter, AIVA is based on the ENT meta-model. Thus it solves problems with CBSE by simply using ENT for description of components and their structure and visualizing information in the ENT, that is general.

First of all, we did not want to create a new visual notation when it is not needed, so we *reused* several principles from UML of how the components should look and how they should be connected, and added several improvements or novel features that UML does not offer. A key difference from from UML is *information hiding* that is bound to how the components are presented. The key idea is to show only what is important at the current level of abstraction. These principles are behind the notation core described in Section 6.3. Moreover, component elements are grouped into logical groups which provide better overview about the characteristics of these elements. As a result components are easily understandable and readable. This grouping is described in Section 6.4.

In diagrams of complex applications, the complexity of connection lines can completely exceed users' cognitive capacity. Such complex diagrams are quite easy to get in this domain of visualization. We propose a substantial reduction of connection lines – as they cause the most of visual clutter. Moreover, other reduction, compared to the UML component diagram, will be acquired by completely eliminating visualization of interfaces as a standalone nodes. Such approach would provide a significant reduction of visual elements and result in a better readable diagram. Interactive techniques will be used to balance the information loss – meta-model behind the diagram will kepp all the information, but diagram will present them only on deman. The reduction of connection lines is described in Section 6.5.

Similarly we propose a reduction of complexity in hierarchical application by simply hiding all the information about the inner structure. Showing only the top level components to get the understanding about the structure more easily. These details will be again accessible through the interaction. These principles are discussed in Section 6.6.

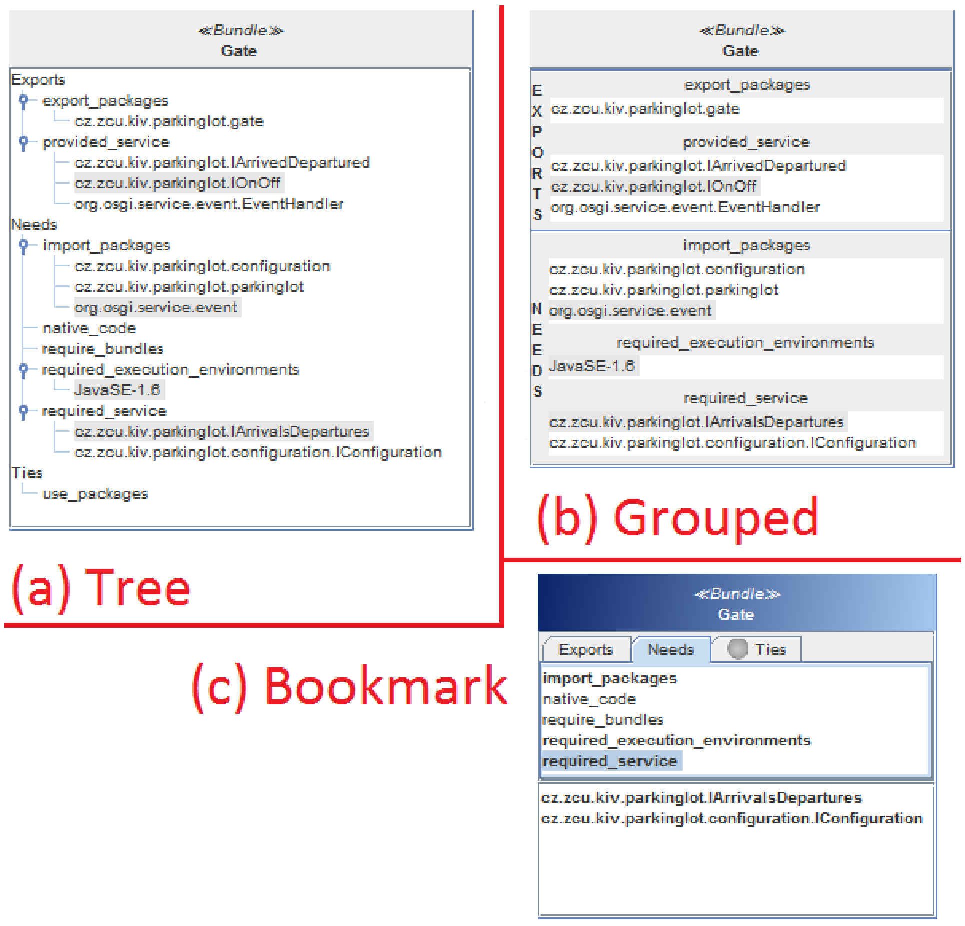
AIVA offer three additional features that will be most helpful for different development roles. First one eliminates the need for multiple diagrams for every role. It is conditional filtering of elements, that will show only the information that is of some interest for the particular role. These conditional filtering rules can be designed by users themselves, thus it is guaranteed that they will see only what they need. These rules use the ENT classification, thus the filtering is semantic. It is based on features from the ENT meta-model and it is described in Section 6.4.

Second feature called "Structure mode" addresses the needs of system architects and assemblers. It is designed to help working with the whole structure, providing only the most elementary information, while not losing all the advantages of our proposed visualization technique. Therefore it has a quick access to all details. This is briefly described in Section 6.7.

Last feature is conditional highlighting. All the components in diagram look similar by design, making it harder when one needs to focus on one specific group of them. However these components can be easily highlighted by creating conditions with user defined rule sets. The different groups of components can be tracked easier – thus the whole structure is better readable. Depending on the quality and purpose of user defined conditions the learning process can be significantly faster. Details concerning conditional highlighting are in Section 6.8.

6.3 Visual Notation of Components

The visual representation of a single component is described here. The header of a component has two lines: the type of component is enclosed by guillemets on the first line and the name of the component is present on the second line. This was inspired by the UML notation. The header is the only things that does not change; the body of the component can be altered as the user needs. Basically there are two ways that change the body of each component – 1) which elements should be visualized (see Section 6.4 for filtering); 2) and how to visualize them (styles of representation).



«Bundle»
Gate

Exports
- export_packages
  - cz.zcu.kiv.parkinglot.gate
- provided_service
  - cz.zcu.kiv.parkinglot.IArrivedDepartured
  - cz.zcu.kiv.parkinglot.IOnOff
  - org.osgi.service.event.EventHandler

Needs
- import_packages
  - cz.zcu.kiv.parkinglot.configuration
  - cz.zcu.kiv.parkinglot.parkinglot
  - org.osgi.service.event
- native_code
- require_bundles
- required_execution_environments
  - JavaSE-1.6
- required_service
  - cz.zcu.kiv.parkinglot.IArrivalsDepartures
  - cz.zcu.kiv.parkinglot.configuration.IConfiguration

Ties
- use_packages

«Bundle»
Gate

EXPORTS
export_packages
cz.zcu.kiv.parkinglot.gate
provided_service
cz.zcu.kiv.parkinglot.IArrivedDepartured
cz.zcu.kiv.parkinglot.IOnOff
org.osgi.service.event.EventHandler

NEEDS
import_packages
cz.zcu.kiv.parkinglot.configuration
cz.zcu.kiv.parkinglot.parkinglot
org.osgi.service.event
required_execution_environments
JavaSE-1.6
required_service
cz.zcu.kiv.parkinglot.IArrivalsDepartures
cz.zcu.kiv.parkinglot.configuration.IConfiguration

«Bundle»
Gate

Exports | Needs | Ties

import_packages
native_code
require_bundles
required_execution_environments
required_service

cz.zcu.kiv.parkinglot.IArrivalsDepartures
cz.zcu.kiv.parkinglot.configuration.IConfiguration

Figure 6.1: OSGi component called “Gate” in all representation styles used by AIVA.

Figure 6.1 contains one OSGi component called “Gate” represented using various styles. All these styles are implemented by AIVA and are supposed to offer different experience based on user’s preferences. All these representations use similar grouping mechanism – Category Sets (see Section 5.3.5). Every element is grouped based on its type, which is also applied by UML. However AIVA uses yet another level of grouping based on shared characteristics of these types of elements. For example: `cz.zcu.kiv.parkinglot.IArrivedDepartured` is a provided service, thus it is in group `provided_service`. All provided services and exported packages are exported/provided by component "Gate" thus they are grouped under `Exports` main group.

A single element is displayed in the classical way as *nameOfElement: type*. If it does not have any type defined (and also when the type is not important or is always the same), it is displayed only as *nameOfElement*. Types of elements are used, e.g., with CORBA components (e.g. Figure 6.2), unlike for OSGi, where elements are the names of interfaces, classes and packages (e.g. Figure 6.1).

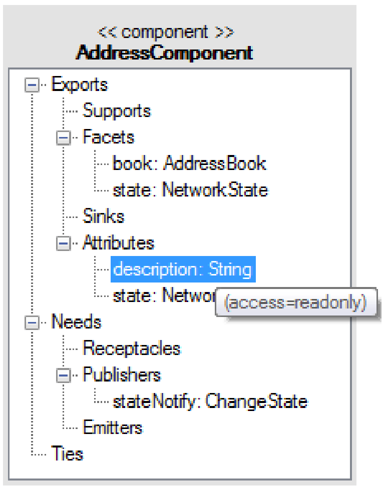


Figure 6.2: How would AIVA represent a CORBA component.

Tree representation is quite different from the UML. It presents the above described grouping in a tree structure, which is the most logical way how to present this type of information. This representation shows all groups, even the empty ones. The advantage of this representation is, that hierarchy is apparent on the first look. The disadvantage is that elements coincide with group names.

Please note that every representation style presents components from different component models in the same manner, compare CORBA component in Figure 6.2 with OSGi component in Figure 6.1(a). Both components are represented in the same way so it is easy to read components from any component model.

Grouped representation is the most similar to the UML one. It contains similar element listings like UML does, however it indicates their main group on the left edge of component. This representation ignores all empty groups and does not bother user with their presence. The advantage is in clear distinction of element, type group and main group. This representation is not so oriented on hierarchy, which may be seen as a disadvantage.

Bookmark representation is the most innovative one. It is inspired by Telea's [66] component browser. It uses bookmarks on the top part of the components to show only its subgroups. Elements are listed in the bottom part of component rectangle only when their parent group is selected. This representation shows all groups, even the empty ones. This representation is the most compact and is the best readable as it only shows the selected information. It should be used for applications composed from very complex components where it will show only the requested information. On the other hand it is quite useless for visualization of small components, where one wishes to see the whole inner structure all at once.

6.3.1 Interactive Features of Component Notation

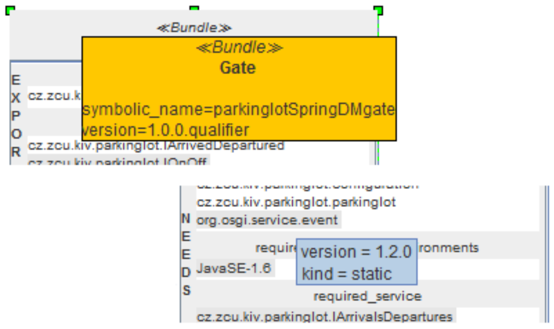

≪Bundle≫
≪Bundle≫
Gate
symbolic_name=parkinglotSpringDMgate
version=1.0.0.qualifier
EXPOR
cz.zcu.k
cz.zcu.kiv.parkinglot.IArrivedDepartured
cz.zcu.kiv.parkinglot.IOnOff
cz.zcu.kiv.parkinglot.parkinglot
org.osgi.service.event
NEEDS
requir
version = 1.2.0
kind = static
ronments
JavaSE-1.6
required_service
cz.zcu.kiv.parkinglot.IArrivalsDepartures

Figure 6.3: Two info boxes used interactively

The ENT meta-model is able to hold more information than the names and types of components and elements. These additional information can say more about concrete component or element, for example: vendor of component, version range, filter of service or else. Such information can be easily accessed directly in the diagram.

To get more details about component, one has to simply click on the header of the component. An info box will immediately appear showing all information available for this component. A duplicate information about component name and type is also included, however it is extremely useful when the diagram is zoomed out and the header of component is otherwise unreadable. Component info box is in the top of Figure 6.3 and additionally it shows the list of all "Tags" (see Section 5.3) used by this component. All tags are listed in following syntax: *tagName=tagValue*. Unused tags are not visible as they lack meaning.

The info box providing more information about single element is accessed by simple hovering with mouse cursor over the element. This info box is shown in the bottom of Figure 6.3 and it only shows the list of all “Tags” used by this element. The name of element is not required in this case, because elements are accessible only when they are readable. This info box uses identical syntax for listing tags as component info box does; unused tags are not listed.

6.4 Diagram Filtering by Category sets

Different users and roles need, in different situations, to emphasize and/or hide some traits and elements. For example, component architects are interested in other information than component developers. By displaying all information contained in the model, on the other hand, there could be a danger of confusion when representing big and complex applications.

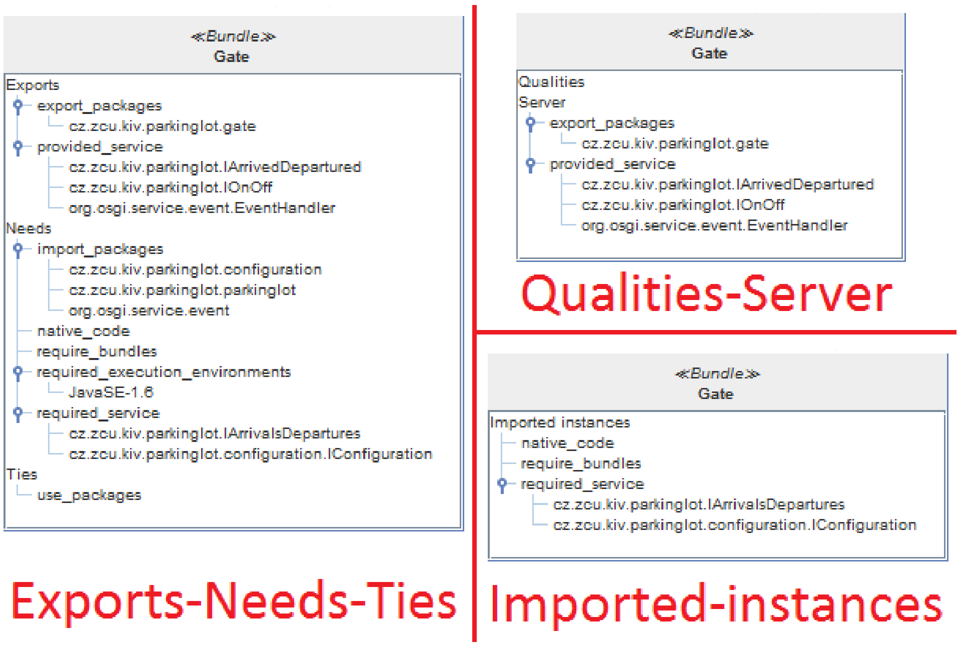

```
«Bundle»
Gate
Exports
  export_packages
    cz.zcu.kiv.parkinglot.gate
  provided_service
    cz.zcu.kiv.parkinglot.IArrivedDepartured
    cz.zcu.kiv.parkinglot.IOnOff
    org.osgi.service.event.EventHandler
Needs
  import_packages
    cz.zcu.kiv.parkinglot.configuration
    cz.zcu.kiv.parkinglot.parkinglot
    org.osgi.service.event
  native_code
  require_bundles
  required_execution_environments
    JavaSE-1.6
  required_service
    cz.zcu.kiv.parkinglot.IArrivalsDepartures
    cz.zcu.kiv.parkinglot.configuration.IConfiguration
Ties
  use_packages
```
```
«Bundle»
Gate
Qualities
Server
  export_packages
    cz.zcu.kiv.parkinglot.gate
  provided_service
    cz.zcu.kiv.parkinglot.IArrivedDepartured
    cz.zcu.kiv.parkinglot.IOnOff
    org.osgi.service.event.EventHandler
```
```
«Bundle»
Gate
Imported instances
  native_code
  require_bundles
  required_service
    cz.zcu.kiv.parkinglot.IArrivalsDepartures
    cz.zcu.kiv.parkinglot.configuration.IConfiguration
```

Figure 6.4: Filtering ENT visualization by category sets.

These problems are solved by using category sets described in Section 5.3.5. These category sets can filter and group traits and then be used to provide the hierarchical structure described in the previous section. There are three different views of the same OSGi bundle in Figure 6.4 to present the possibilities of category sets. These different views will be discussed to suggest the importance and usage of this feature.

The first view is called E-N-T (Exports-Needs-Ties) and it shows all traits of the bundle and group them depending on their role – if they are exported, imported or create ties between export and import. This type of view is valuable for full feature overview and easier orientation in what this component exports and imports.

The second view is called Q-S (Qualities-Server) and it filters a lot of traits away. This view is focused on "server" side of any component and thus it shows only what this component offers – exports. Moreover these traits are grouped to two groups, "Server" stands for functional features and "Qualities" are nonfunctional or semantic characteristics. Such view provides only "server" side on our example because there are no semantical or nonfunctional traits identified for OSGi framework in [67]. In any event, this view is interesting whenever one needs to study only what each component have to offer and their requirements are unimportant.

Finally the last view presented is called II (Imported-Instances) and it also filters a lot of traits away. It is focused on the requirements side of any component, because it shows only what is imported by that component. Moreover it only shows the instances – elements that are instantialized in run time. Such view will help the component assemblers to study the dependencies in run time.

The possibilities of grouping and filtering are very rich, as they can use more than one condition (used by S-Q and II) and define 1-N groups. AIVA contains four predefined category sets that are ready for use and additional sets can be defined by a user. The visualization of an application can thus be parametrized (modified) to suit individual unforeseen needs or to specific roles. Category sets are always applied on the whole diagram, therefore all components changes its body to the new hierarchy structure.

6.5 Inter-Component Bindings

Bindings between two components are represented by a connection line with a "lollipop" notation on it. This style was chosen as it is a standard way introduced by the UML and thus developers can identify it easier. However there is a big difference between "lollipop" in the UML and in the AIVA. UML uses it as a representation of one interface, creating new graph node in component diagram for every interface – making the final diagram more complex. AIVA uses it only to show the direction of relation (provided/required), "lollipop" is then just an image making it easier for users to understand what is provided and what is required.

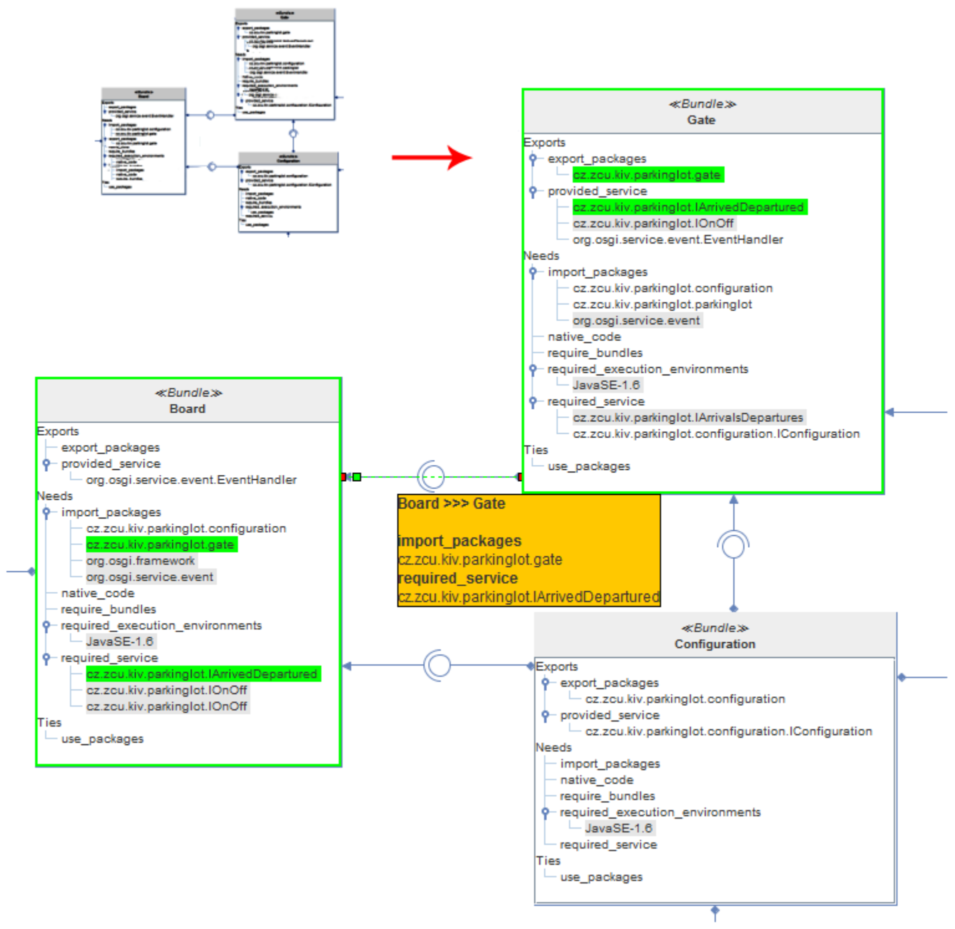

«Bundle»
Gate
Exports
export_packages
cz.zcu.kiv.parkinglot.gate
provided_service
cz.zcu.kiv.parkinglot.IArrivedDepartured
cz.zcu.kiv.parkinglot.IOnOff
org.osgi.service.event.EventHandler
Needs
import_packages
cz.zcu.kiv.parkinglot.configuration
cz.zcu.kiv.parkinglot.parkinglot
org.osgi.service.event
native_code
require_bundles
required_execution_environments
JavaSE-1.6
required_service
cz.zcu.kiv.parkinglot.IArrivalsDepartures
cz.zcu.kiv.parkinglot.configuration.IConfiguration
Ties
use_packages
«Bundle»
Board
Exports
export_packages
provided_service
org.osgi.service.event.EventHandler
Needs
import_packages
cz.zcu.kiv.parkinglot.configuration
cz.zcu.kiv.parkinglot.gate
org.osgi.framework
org.osgi.service.event
native_code
require_bundles
required_execution_environments
JavaSE-1.6
required_service
cz.zcu.kiv.parkinglot.IArrivedDepartured
cz.zcu.kiv.parkinglot.IOnOff
cz.zcu.kiv.parkinglot.IOnOff
Ties
use_packages
Board >>> Gate
import_packages
cz.zcu.kiv.parkinglot.gate
required_service
cz.zcu.kiv.parkinglot.IArrivedDepartured
«Bundle»
Configuration
Exports
export_packages
cz.zcu.kiv.parkinglot.configuration
provided_service
cz.zcu.kiv.parkinglot.configuration.IConfiguration
Needs
import_packages
native_code
require_bundles
required_execution_environments
JavaSE-1.6
required_service
Ties
use_packages

Figure 6.5: Part of a simple OSGi application after the user requested the information.

This might feel a little confusing, however AIVA treats all the component elements equal – all of them are represented in the same way, thus interfaces alone should not be pictured differently. This uniform representation allows AIVA to collapse all the connection lines between two components into one and only line. Such line then represents the relation "component A provides 1-N elements to component B" instead of the commonly used "component A provides interface X to component B". This feature is yet another way how to decrease the number of graphical elements and thus increase the readability of the final diagram.

Moreover AIVA does not add any information label as UML does because it would again only increase the number of graphical elements. All these features might feel as an disadvantage, however because of AIVA interactivity it is very easy to provide these information only when user needs them – diagram just does not need to bother user with these information all the time. How this interactivity works is shown in Figure 6.5. Top left corner shows the default view on all three components, which is clear without any unwanted information. The main part then shows how the diagram highlights the connection after user click on the connection line, details are discussed in Section 6.5.1.

By studying Figure 6.5 more in depth one can found, that connection lines have arrows on its ends, which might seem indistinct on this figure but it is distinct enough in real application. Diamond shaped arrow means that elements are provided/exported and closed arrow means that elements are imported/required. These arrows yet complements to the information provided by the “lollipop” notation however they are directly readable on the edges of each component. It is then easier to study one component and immediately know which connection lines stands for provided or required elements. It is also easier to keep orientation which lines are connected with that one component and which lines continue elswhere.

Yet another information related to connections and relations is contained in the discussed Figure 6.5. All elements, that are not used to create any connections are emphasized with gray color. This helps users to instantly find unsatisfied requirements, unused exported packages and others. Or from opposite point of view it helps to find elements, that are more interesting for user as they are used.

6.5.1 Interactive Features to Reveal Relations

When a user clicks on the connection line, the information box appears, summarizing all connections between these two components with all elements listed and trait allegiance noted. To make these connected elements more visible they are also highlighted, together with connection line and both components. Component highlighting is highly usable when components are far away – it is easy to find them almost immediately. These components are also highlighted in the overview of diagram making it even more intuitive to locate them in the context of the whole diagram.

When user double click on the element, all components that use it are highlighted in both diagram and overview. To make it even easier for user to find what he is looking for, AIVA also highlights all the connection lines, that connects the parent component of clicked element with all related components. However no additional textual information is shown and user himself must to evaluate what he needs to know.

The described information box and highlighting are the substitution for persistent presence of all graphical elements known from the UML. The approach has simply a different philosophy than UML, because it puts information that two components are related on the first place and information through which elements comes after that.

6.5.2 Element Oriented Connections

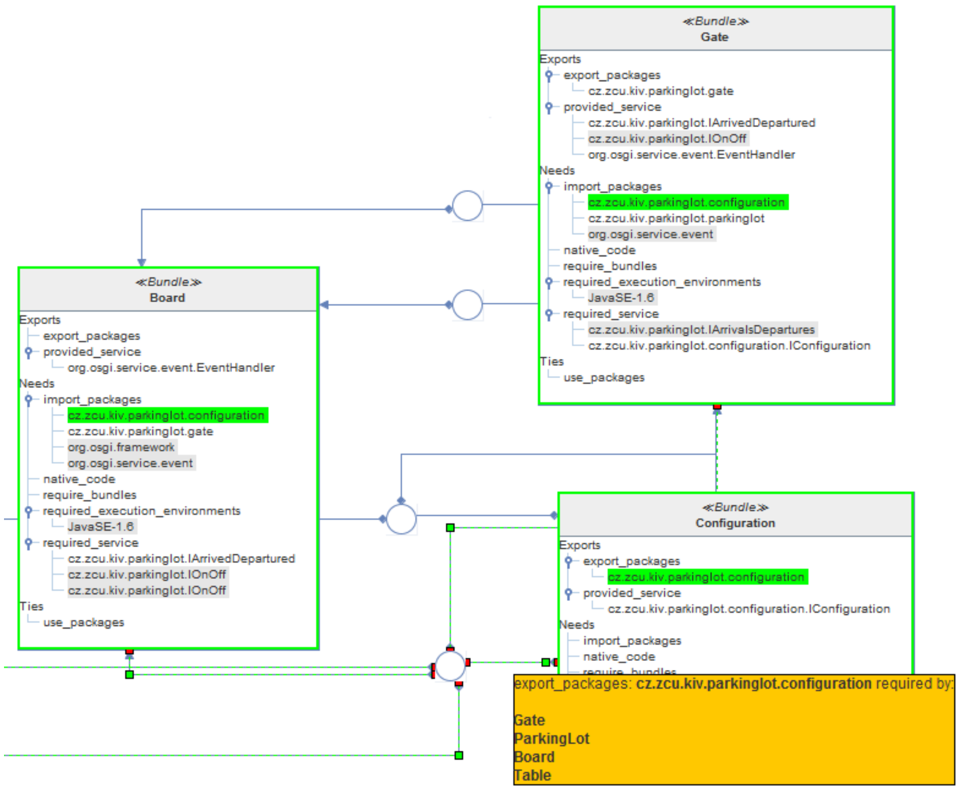

«Bundle»
Gate
Exports
export_packages
cz.zcu.kiv.parkinglot.gate
provided_service
cz.zcu.kiv.parkinglot.IArrivedDepartured
cz.zcu.kiv.parkinglot.IOnOff
org.osgi.service.event.EventHandler
Needs
import_packages
cz.zcu.kiv.parkinglot.configuration
cz.zcu.kiv.parkinglot.parkinglot
org.osgi.service.event
native_code
require_bundles
required_execution_environments
JavaSE-1.6
required_service
cz.zcu.kiv.parkinglot.IArrivalsDepartures
cz.zcu.kiv.parkinglot.configuration.IConfiguration
Ties
use_packages
«Bundle»
Board
Exports
export_packages
provided_service
org.osgi.service.event.EventHandler
Needs
import_packages
cz.zcu.kiv.parkinglot.configuration
cz.zcu.kiv.parkinglot.gate
org.osgi.framework
org.osgi.service.event
native_code
require_bundles
required_execution_environments
JavaSE-1.6
required_service
cz.zcu.kiv.parkinglot.IArrivedDepartured
cz.zcu.kiv.parkinglot.IOnOff
cz.zcu.kiv.parkinglot.IOnOff
Ties
use_packages
«Bundle»
Configuration
Exports
export_packages
cz.zcu.kiv.parkinglot.configuration
provided_service
cz.zcu.kiv.parkinglot.configuration.IConfiguration
Needs
import_packages
native_code
require_bundles
export_packages: cz.zcu.kiv.parkinglot.configuration required by:
Gate
ParkingLot
Board
Table

Figure 6.6: Connection representation that emphasizes elements.

The last feature that helps to reveal relations between components is different representation of connections. Although we present the component oriented representation at the first place, there are cases when interface oriented representation, known from the UML, might be more beneficial. All exported elements are equal thus they have their own "lollipop" symbol, which is used only for interfaces in UML. This alternative representation clearly does not benefit from specific features described earlier. However it still introduces some level of interactivity.

When user switches between connection representation, all connection lines are replaced to conform to this different design. However "lollipops" are not real graph nodes, they are rather considered to be a part of component, that provides them and are pictured right next to its parent component. Trying to make the diagram a little cleaner AIVA does not show labels next to “lollipops”. However, the number of connection lines is not reduced. This different representation is in Figure 6.6 all at once with interactivity demonstration. When the “lollipop” is clicked it shows the information box containing the name of the connected elements with the list of components, that uses it. All connected components and corresponding connection lines are highlighted for easier orientation. Overview again shows all these highlights.

6.6 Composite Components

The structure of component-based applications becomes complicated when higher-level components use other composite (sub)components. The level of recursion can be rather high, thus making the diagram, where all these composite components show their internal structure, hard to read and understand. The key to lower this complexness is in hiding of the inner structure of these composite subcomponents, which should still be easily accessible. This is the reason why AIVA displays composite components similarly to atomic components, without revealing their internal structure. It only informs user that inner architecture is present by the key word *composite* in the upper right corner of the component.

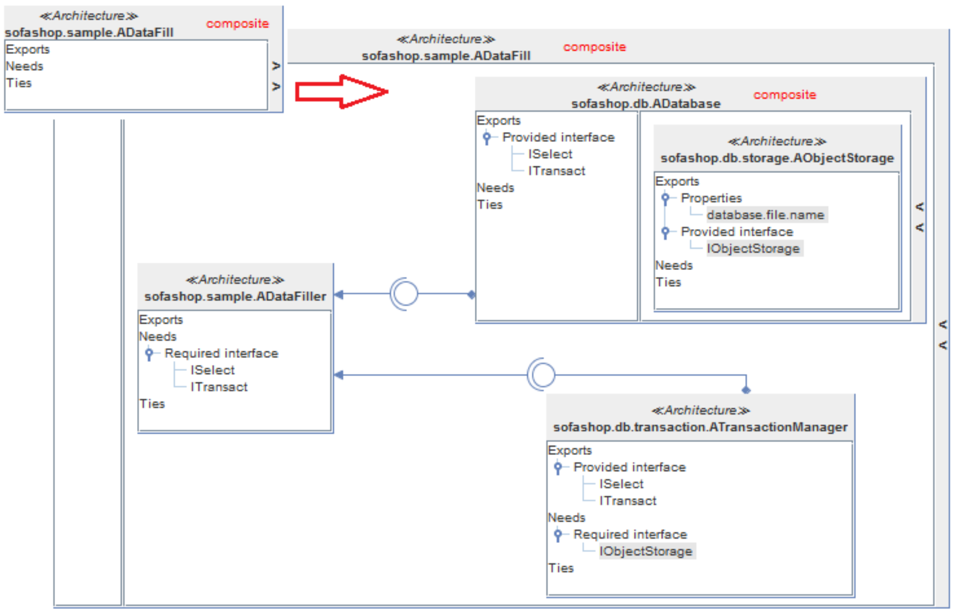


Figure 6.7: The expand principle of composite components.

The user can study an overall structure of the application without any disturbances caused by its composite subcomponents, however he is aware of them. When he decides that he is ready to study the inner structure of these composite components he can open it one by one as he needs. The component box expands itself unveiling the detailed view on the inner structure by using the expansion arrows along the right edge of the composite component. Figure 6.7 shows both expanded and collapsed (top left corner) component. The detailed view is another full featured diagram view on the inner structure, that is encapsulated in the body of its parent component. This feature keeps the diagram of the hierarchical application simple, because every expanded component can be collapsed again, and does not require the creation of any other separate diagrams to study the inner structure of composite components.

Some cases may however benefit from the possibility to visualize the inner structure in the different view. These cases may be for example – SOFA uses one main component to encapsulate the structure of the whole application; some composite components can have just too complex inner structure to be visualized expanded. AIVA makes it easy to create these different views though. Right mouse click on any composite component open the context menu, that offers to open its inner structure in new tab. The content is automatically created and does not require any additional tasks. This view uses full featured AIVA and offers the same functionality as the original diagram.

6.7 Simple Structure View

Component assemblers need most of the time to see only the overall structure of the whole application, but they might need to study the details of the component to check the compatibility and substitutability. Component developers are on the other hand interested in the details most of the time, but when they need to find related components they often zoom out to find these components more easily, however the name of component is then unreadable, because it is too small.

Therefore, the structure view presents all components with the body part of the box hidden, so all that remains from the component representation are the names of the components and their types in guillemets plus the connection lines (see Figure 6.8). This results in a clean and simple "boxes-and-arrows" diagram. This view is activated immediately after the user zoom out. The component name and type are readable, because AIVA automatically sets higher font size, when this view is activated. The transition between detailed view and structure view is seamless to offer the maximum user experience. Interactivity still allows to click on any component and get

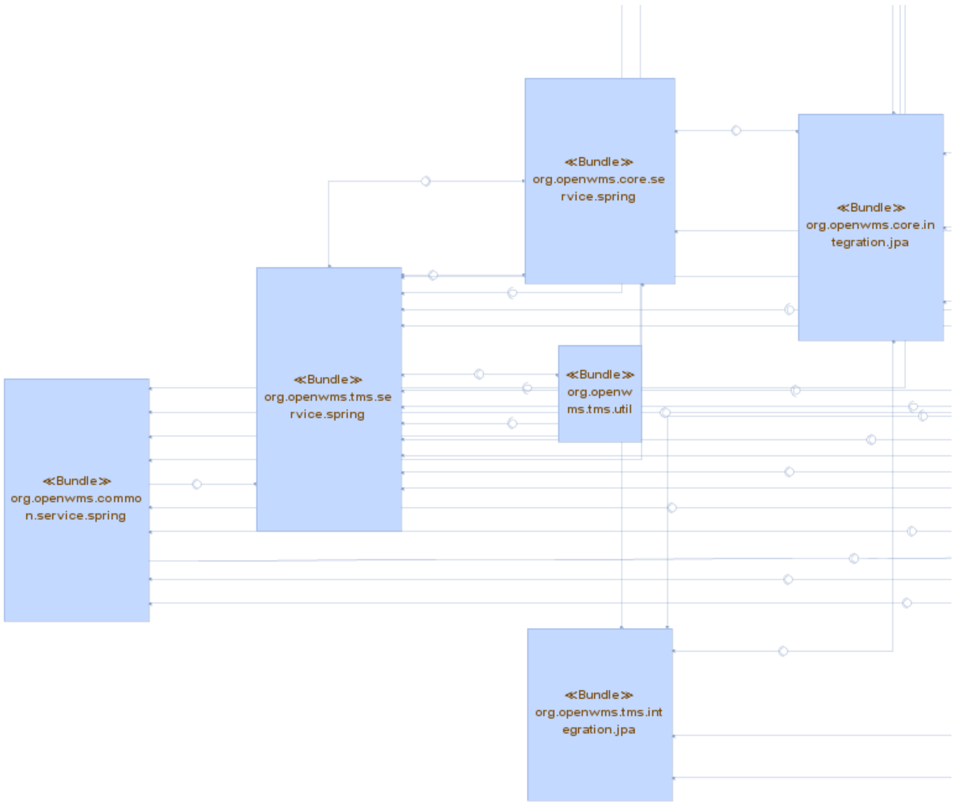


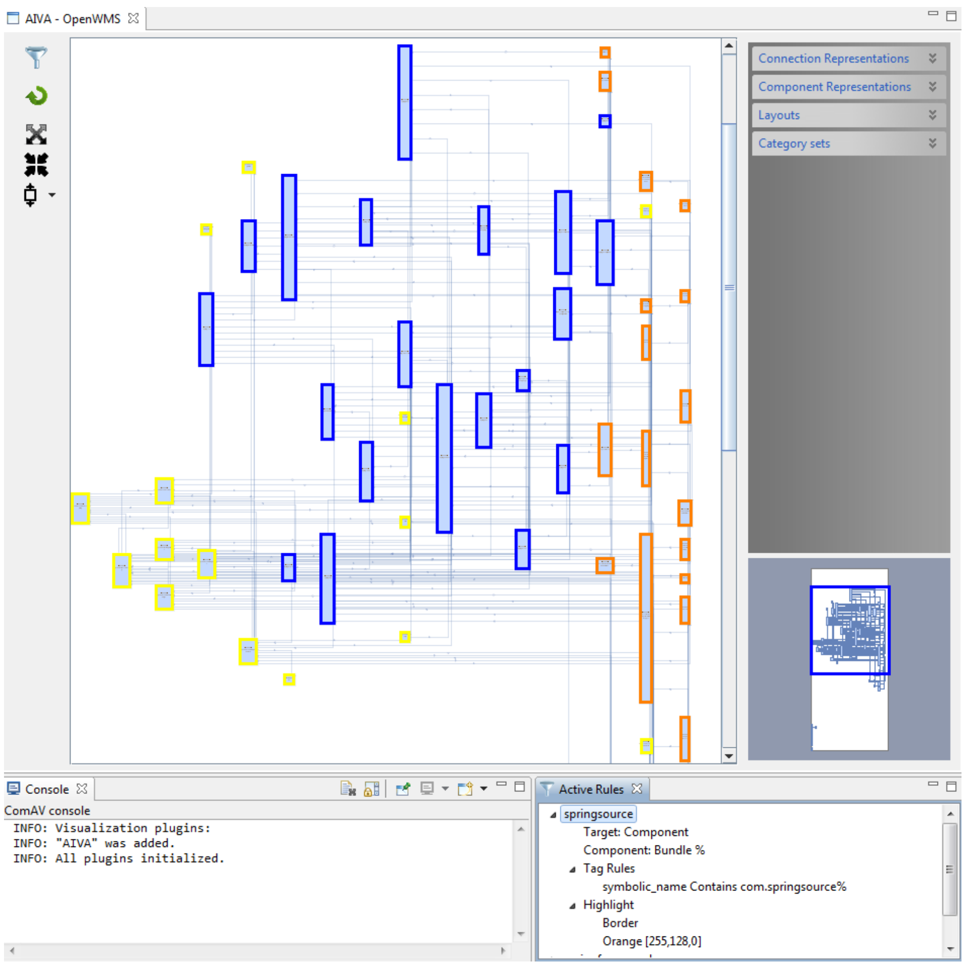
Figure 6.8: Simple structure view over several components

readable info box with all the details.

6.8 Conditional Highlighting

Highlighted information is easier to find in large amount of data. When working with dozens of different components in a complex graph structure any additional visual information helping user to find what he is looking for is valuable. Conditional highlighting is an excellent example of such feature that add just another visual information based on specified conditions. If user could create conditions that conform to his current needs and apply them on the whole diagram, it would increase his performance.

The ENT structure, that is behind AIVA can contain really large amount of data, that are commonly hidden. Generally these data are stored in the "Tag" structure, completing the information about component or its elements. All these information are accessible upon interaction however when one needs to find components or elements, that have some special settings



```
INFO: Visualization plugins:
INFO: "AIVA" was added.
INFO: All plugins initialized.
```

```
springsource
    Target: Component
    Component: Bundle %
    Tag Rules
        symbolic_name Contains com.springsource%
    Highlight
        Border
        Orange [255,128,0]
```

Figure 6.9: OpenWMS application highlighted with different conditions.

it might be quite frustrating. Conditional highlighting is an excellent way how to emphasize some of these information if needed, or to enhance the orientation while highlighting different component groups.

Figure 6.9 shows an example usage of conditional formatting. The visualized application is assembled by using components from three different providers. Each provider is highlighted by a different color at the border of each component. The condition compares "symbolic_name" value, that is saved for each OSGi bundle. This case shows how conditional highlighting could improve orientation in complex diagram structures by highlighting different component groups.

Conditional highlighting in AIVA is based on these ideas (some related to the ENT meta-model):

1. It can highlight both components and their elements, because both of them can have some secondary information.
2. Element highlighting can be transfered on the whole component - so it is easier to find what one is looking for.
3. Rules are component model specific so they can be well suited for specific needs.
4. Rules can be set to specific component types and specific traits if needed. Which means only these will be analyzed and highlighted and the rest of components/elements will be avoided.
5. Additionally one can create complex rules by adding requirements on component/element tags. All tag rules have logical AND operator thus creation of very strict rules is possible.
6. Logical OR between rules can be achieved by creating similar new rule, with identical highlighting settings.
7. Tags can be tested if they contain some value, or if its value is in predefined range.
8. One can choose how he wants to highlight defined condition (set of rules). For example: persistent thick border or colored component background.
9. One can always choose which conditions he wish to use and create own subsets of conditions that are best suited for ones needs.

It should be emphasized that this highlighting is component model specific. Otherwise it would not be possible to offer anything more intelligent than component/element name based highlighting. It is important to create an editor, that will be able to work with all these ideas and which will only load and save conditions related to the relevant component model.

6.9 Other Features

Other features of AIVA are rather standard in visualization of software or are not so important for the AIVA. However, they are worth mentioning and therefore are analyzed in this Section.

The navigation in the diagram was not discussed yet, however scrolling is somehow automatic feature enabling user to explore the diagram. In Section 6.7 we already mentioned zooming technique, that has a special meaning, however when the details are hidden it behaves as expected. To complete the set of navigation techniques AIVA is supported by panning – enabling simple moving over short distances.

Although the overview was not mentioned in previous paragraph it is present. It was several times mentioned through different sections in this chapter, because it has a great impact on the overall orientation. It enables the classical navigation and provides a brief view of the whole diagram, moreover it is able to highlight a lot of information – as they are highlighted in the diagram. User does not have to zoom out to find connected component, for example, but he see it immediately in overview, which can be used for extremely fast navigation and exploration.

The ability of AIVA to change the representation of components and switching between different connection representations was already mentioned. More common change of layout was not discussed yet though. AIVA of course support this feature and is able to change between several different layouts to present the same structure in different arrangement.

Last group of features helps to change visual feeling of the diagram. The tree structure of each component was discussed, such representation might be very spatially demanding given complex components with dozens of elements. Therefore AIVA implements features that give user control over this, so he can easily reduce its height, without the need to filter some elements away. It is possible to collapse tree structure so only "Categories" are visible, it is possible to expand tree structure so all elements are visible. All components can be set on the same height, keeping all elements expanded and prepared for future investigation. When components' height is adjusted it is required to refresh layout, so these "node size" changes are reflected into the arrangement. Finally it is possible to change colors and styles so users will feel more comfortable.

6.10 When It Is Better To Use AIVA Than UML

Let us conclude this section with a brief discussion of the situations where it is better to use UML component diagrams and when it is better to use our interactive visualization, because each of them is best for different kinds of things. The situations in which the two visualization alternatives were compared are presented in Table 6.1. This comparison assumes that UML extended with a profile is used, to provide the similar information, so these two approaches are comparable.

Table 6.1: Comparison of visualizations.

Situation	*ENT*	*UML*
User needs to create a high-level mental model from the diagram(s)	X	
Application has to be described on several levels of details	X	
User needs to work on several levels of details seamlessly	X	
Dynamic aspects of the application need to be modeled		X
Application with many components and connections	X	
Diagram is presented on paper		X
User needs to present a diagram in a generally known format		X

Chapter 7

Component Application Visualizer

Component Application Visualizer (ComAV) is a universal platform for visualization and reverse-engineering of component-based software. The purpose of ComAV is to create a generic workspace for visualization and management of analyzed applications. This workspace provides only the basic management functionality and most importantly it is general – it is not bound to any component model or visualization style. Instead it offers two different extension types – loaders, used to reverse-engineer applications; and visualizations, used to visualize these applications. One can add support for new component models, by adding loader plug-in and vice versa for new visualization styles. ComAV manages the exchange mechanisms between these plug-ins and allows them to extend the core – add new menu items, add new views, etc. Plug-ins can not work without ComAV and ComAV have no meaning without plug-ins – at least one loader and visualization plug-in is required to offer some functionality.

The ENT meta-model is used as exchange and storage data structure. Its advantages were discussed in previous Chapter but most importantly it is general and it can hold enough details required for thorough description of structure.

ComAV was developed completely as part of this work and it is licensed under GNU-GPL thus open-source. It provides a good basis for future work on different visualization styles as it is so easily extend-able. It was managed using Assembla [1] tracking system, which also provides public access to its source code.

[1]http://www.assembla.com/spaces/comav

7.1 Conceptual Structure

The origin of principles and ideas behind ComAV is in the ENT meta-model itself.

The ENT is general, but any component model needs to be described prior its use in the ENT. General reverse-engineering can not be possible for component-based applications, due to big differences between component models. Thus implementation of the ENT would either have to add support for all component models or create some extension mechanism. New component models and reverse-engineering support would be added when needed – so it could really support any component model.

The ENT is able to describe any component in detail, its information is categorized, etc. Thus it can be used by more approaches than the one described in this paper – AIVA. Analytical visualizations can make a great use of its content awareness. Therefore the reverse-engineering and management parts can be either reused in different visualization projects or an extension mechanism will be provided. So new visualization approaches can be added next to the existing ones.

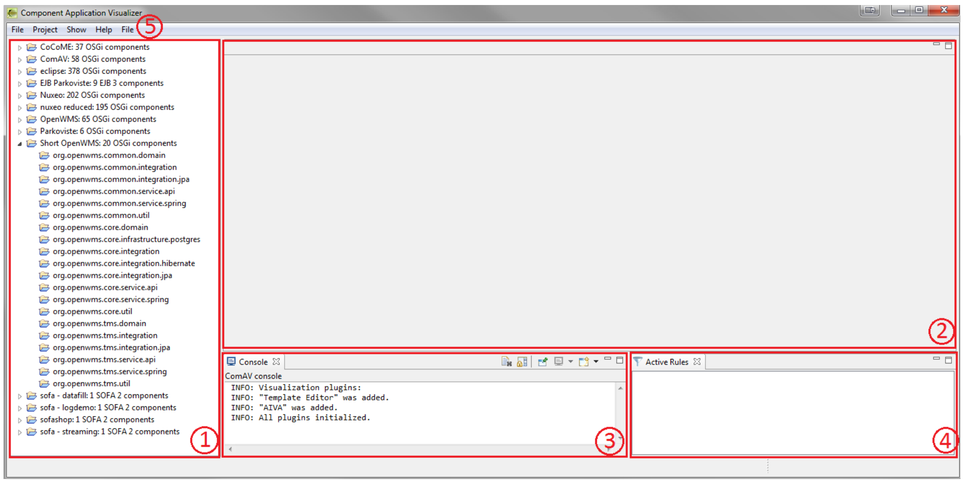
The final design of ComAV is based on these two criteria and its advantages are in short:

1. Any component model reverse-engineering tool can be added.
2. Any new visualization approach can be added.
3. User has one workspace with access to all component models and visualization approaches.
4. By adding a new component model support one can immediately visualize new structures in any visualization approach.
5. One application can be visualized by several approaches. The user have therefore choice which one to select.
6. Several structures and visualizations can be opened at once, thus comparing is possible.

7.2 ComAV GUI

ComAV uses classic project driven environment, known for example from Eclipse IDE [2]. Composition of Comav workspace is introduced in Figure 7.1.

[2]http://eclipse.org/



```
CoCoME: 37 OSGi components
ComAV: 58 OSGi components
eclipse: 378 OSGi components
EJB Parkoviste: 9 EJB 3 components
Nuxeo: 202 OSGi components
nuxeo reduced: 195 OSGi components
OpenWMS: 65 OSGi components
Parkoviste: 6 OSGi components
Short OpenWMS: 20 OSGi components
    org.openwms.common.domain
    org.openwms.common.integration
    org.openwms.common.integration.jpa
    org.openwms.common.service.api
    org.openwms.common.service.spring
    org.openwms.common.util
    org.openwms.core.domain
    org.openwms.core.infrastructure.postgres
    org.openwms.core.integration
    org.openwms.core.integration.hibernate
    org.openwms.core.integration.jpa
    org.openwms.core.service.api
    org.openwms.core.service.spring
    org.openwms.core.util
    org.openwms.tms.domain
    org.openwms.tms.integration
    org.openwms.tms.integration.jpa
    org.openwms.tms.service.api
    org.openwms.tms.service.spring
    org.openwms.tms.util
sofa - datafill: 1 SOFA 2 components
sofa - logdemo: 1 SOFA 2 components
sofashop: 1 SOFA 2 components
sofa - streaming: 1 SOFA 2 components
```

```
INFO: Visualization plugins:
INFO: "Template Editor" was added.
INFO: "AIVA" was added.
INFO: All plugins initialized.
```

Figure 7.1: ComAV workspace

Most important part is a project view (1), containing the list of all loaded projects. Every project stands for one analyzed application in any component model, because ComAV is general. This list provides the following information: name of the project, number of components and original component model – for easier orientation. The comprehensive list of all components that create the application is also provided as a subtree to provide full overview and to speed the search process in diagram.

This project overview is used to start visualization in editor view (2). One can use double click on "'project name"' for previously used visualization style or right click "'project name"' to access context menu and choose any visualization style available. Double click on component in project view will center the diagram in editor view on selected component. These features makes use of different visualization styles and navigation easier.

Console view (3) is used for information purposes as all system messages are written there. Active rules view (4) is added by plug-in and demonstrate the ability of plug-ins to integrate with ComAV. This view is used to list active rules used for conditional formatting in diagram. Finally menu (5) is used to create new projects. Loader plug-ins can extend menu to add support for new component models. It also contains commands that helps with management of listed projects like rename or delete.

7.3 Structure of ComAV

ComAV itself is a component application, built from three main components (core, libraries and ENTMM) and at least two plug-in components. The technology used for implementation is a Rich Client Platform (RCP) [31] based on the Eclipse IDE[3]. RCP brings the advantages of Eclipse, such as simple extensibility mechanism, a lot of different components ready for use and a lot more to any new application. RCP's extension mechanism is called extension point and defines several conditions that needs to be met. In general it defines a handler, that is required to be implemented by plug-in to be recognized. Such handler is a simple Java class that extends predefined `AbstractHandler` class.

ComAV defines two types of these extension points as suggested before – loader and visualization. Both these extension points will be discussed in further part of this section. The overall structure of ComAV is very simple and consists from these components:

1. **ComAV – core**: a main application window. It contains GUI, defines extension points for plug-ins and implement all project management and workspace functions. It is the most important part of ComAV and new functionality related to ComAV should be added here, or connected through new extension points.
2. **ComAV – libraries**: a pack of common libraries used by most of ComAV plug-ins. These libraries are provided through `Export␣Package` functionality of any OSGi bundle. This component is not connected through any extension point, it passively provides packages for the rest of components.
3. **ENTMM**: an implementation of the ENT meta-model in form of passive OSGi library. It is also used through `Export␣Package` functionality by every ComAV component, as it is a core library for the whole application. It provides the ENT structure, used for description of both component-based application and component model, it can moreover save and load the ENT structure in a XML file.
4. **JGraphBasic**: a library that build on top of JGraphX[4] graph library. It creates a general graphical elements in JGraph based on the ENT structure and it makes use of JGraph easier. This component is also used by simple `Export␣Package` functionality and it is used only by visualization plug-ins that depends on JGraphX library.
5. **Loader**: there can be several loaders in ComAV, but there should be at least one to provide some functionality. All loaders are connected to core through its extension points and use the ENT meta-model to store the structure of the application. This structure is an output of any loader.

[3]http://www.eclipse.org
[4]http://www.jgraph.com/jgraph.html

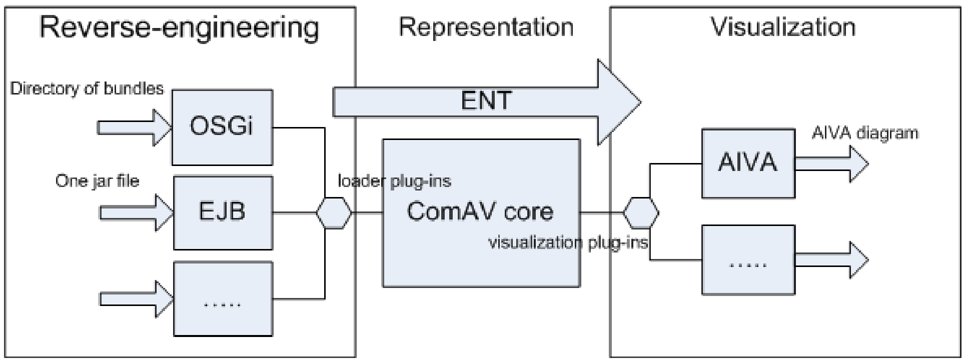


Figure 7.2: Information flow in ComAV application

6. **Visualization**: Similar to loader however the ENT structure is used as an input and plug-ins of this type use it for visualization purposes.

The basic structure is quite apparent from this short but complete list of components. The importance of the ENT meta-model should be also apparent. Both structure and data flow can be read from Figure 7.2. Core is connected only to loader and visualization plug-ins, the rest of components have library purpose only. The ENT meta-model is used as an independent and general exchange format. Loaders reverse-engineer the application so collected information can be stored in the ENT structure. This structure is handled to the core, where project is created, structure saved in workspace and whole project is ready for future visualization. Since then the structure is component model independent and can be visualized in the same manner by any visualization plug-in. To do so a visualization plug-in is called with the concrete ENT structure as an input.

7.3.1 Loader extension point

All loader plug-ins are loaded and recognized at the start up of the ComAV application. All of them extend the user interface with only one menu item – New project. How are loaders integrated into ComAV GUI is shown in Figure 7.3. Loader is component model specific and thus it is only able to reverse-engineer the application of that one specific component model. When new project is started ComAV calls `execute()` method in handler of that specific loader.

The abstract loader handler that is defined in ComAV core is in Listing 7.1. Every loader plug-in has to implement its own handler, that extends this one and add the real functionality – implement `loaderExecute()` method. In the listings one can see only the most important part of the listed class. While `loaderExecute()` method handles the real loading (reverse-engineering),

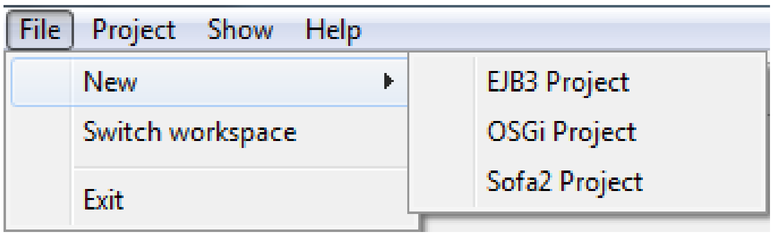


Figure 7.3: Main menu, where new loader plug-ins are added

`execute()` method runs checks if everything was loaded properly – return value is true and all attributes are not null and complete.

The `loaderExecute()` method has two main tasks:

1. Create project wizard: wizard has to ask user at least on two information – project name and how to get the application. Every loader may require different kind of input information to get the application, some loaders might need directory, different loaders might prefer deployable archives or repository address.

2. Reverse engineer: with the informations from wizard, loader is finally able to find and analyze provided application.

At the end, all acquired information is handled to the ComAV core in `execute()` method and loader finishes.

Listing 7.1: CommonLoaderHandler – extension format for all loader plug-ins

```
public abstract class CommonLoaderHandler extends AbstractHandler {
    protected ComponentModel compModel;
    protected List<Component> componentList;
    protected String projectName;
    protected String location = null;

    /**
     * Method called through <b>Loader</b> <code>MenuContrubution</↩
         code>'s
     * handler in core of application.
     * This method calls loaderExecute and handles its return values↩
          and
     * exceptions. It also creates project and handles it to the ↩
         core.
     */
    @Override
    final public Object execute(ExecutionEvent event) throws ↩
        ExecutionException {
      ...
      ...
    }
```

```
        /**
      * <b>Loader</b> plug-ins must loads ENT
      * model and component list in this method.
      *
      * @param event
      *            execution event
      * @return true if loading was completed, false otherwise (user ↩
        pressed
      *         cancel or something)
      * @throws ComAVException
      *             if error during <code>ENT</code> model loading ↩
        error occurs
      */
    abstract public boolean loaderExecute(ExecutionEvent event)
            throws ComAVException;

    ...
    // other getters and setters
    ...
}
```

7.3.2 Visualization extension point

All visualization plug-ins are loaded and recognized at the start up of the ComAV application, similarly like loader plug-ins. However visualization plug-ins extend context menu rather than main menu. This context menu is shown in Figure 7.4 and it is initialized when user right click on project in project view. Visualization is component model independent and thus it can open any ComAV project for visualization. This visualization is initialized by choosing from the context menu or by double clicking on the project name, double click will use default or last used visualization. After this interaction the `displayEntModel()` method is called, which must be implemented by visualization plug-in.

Visualization plug-ins must extend handler listed in Listing 7.2. The `display-EntModel()` method is the most important as it initializes the whole visualization. Plug-ins have to create a new editor window, set correct input data and open it in this point. Note that input data must be set before calling this method, the format can be read in Listing 7.3. It generally uses the same information as loader plug-ins, because it needs project name to label different visualization tabs and component model and list for visualization purposes.

Another important method is `centerOnComponent()` that is either able to open visualization and center it on component, or only to center it on component if the visualization is already open. This method is used for fast navigation between components based on their list in project view (see Section 7.2 for more information). The `getEditorID()` is then used to obtain ID of the concrete visualization editor so the ComAV can automatically reopen last editor used

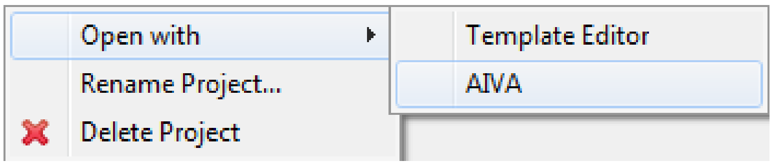


Figure 7.4: Context menu, where new visualization plug-ins are added

automatically. Or to save last used editor settings for each project.

The tasks of implemented visualization handler are more straightforward than the loaders. As already mentioned it only has to create an editor, set proper input and open it. The most of this work is handled by a RCP mechanism, leaving only easy settings on the programmer.

Listing 7.2: CommonVisualizerHandler – extension format for all visualization plug-ins

```
public abstract class CommonVisualizerHandler {
    protected VisualizationEditorInput input;

    /**
     * Called after particular action in core plug-in is called to ↩
         show ENT
     * model in visualizer plug-in.
     *
     * @throws ComAVException
     *              is some error occurs
     */
    abstract public void displayEntModel() throws ComAVException;

    /**
     * Is called by core to center the view on provided component
     *
     * @param component
     */
    abstract public void centerOnComponent(Component component);

    /**
     * Called after particular action in core plug-in is called (i.e↩
         . deleting
     * project). <code>Visualizer</code> plug-in should clear views.
     *
     */
    abstract public void disposeCurrentModel();

    /**
     * Called to get ID of this editor
     *
     * @return editor id
     */
    abstract public String getEditorID();

    ...
    // other getters and setters
    ...
}
```

Listing 7.3: Input class for all editors

```
public class VisualizationEditorInput implements IEditorInput, ↩
    IPersistableElement {

    private ComponentModel componentModel;
    private List<Component> componentList;
    private String projectName;

    public VisualizationEditorInput(ComponentModel componentModel, ↩
        List<Component> componentList, String projectName) {
        this.componentModel = componentModel;
        this.componentList = componentList;
        this.projectName = projectName;
    }

    ...
    // other getters, setters and interface related methods
    ...
}
```

7.4 Supported Component Models

ComAV currently has plug-ins for two commercial component models – OSGi and EJB 3, and one research component model – SOFA 2. OSGi and EJB were chosen to illustrate the usage of tool on wide spread component models and SOFA was selected to show features of hierarchical component model. The specifics of loaders for each component model are covered in this section, concrete ENT specification for these component models can be found in Appendix B. This section will explain, on reasonable level of detail, following topics: which information is gathered, from which source it is taken and the method used to obtain it.

7.4.1 OSGi

OSGi loader requires a directory as an input. This directory has to contain the complete application – all bundles (components) from which it is assembled from. Loader then analyze them all one by one and at the end it evaluate their relations. Most of the information is stored in a bundle manifest, which is a descriptive file, mandatory for every OSGi bundle, saved in `META-INF/MANIFEST.MF`. The structure is predefined and an example manifest file is in Listing 7.4. It describes the bundle itself by using attributes like: `Bundle-Name`, `Bundle-SymbolicName`, `Bundle-Description`, `Bundle-ManifestVersion`, `Bundle-Version`. It also describes the ties of bundle on environment and other bundles by using attributes like: `Export-Package`, `Import-Package`, `Require-Bundle`, `Bundle-RequiredExecutionEnvironment`.

Listing 7.4: OSGi manifest file format

```
Bundle-Name: Hello World
Bundle-SymbolicName: cz.zcu.kiv.loader.osgi.helloworld
Bundle-Description: A Hello World bundle
Bundle-ManifestVersion: 2
Bundle-Version: 1.0.0
Bundle-Activator: cz.zcu.kiv.loader.osgi.helloworld.Activator
Export-Package: cz.zcu.kiv.loader.osgi.helloworld
Import-Package: org.osgi.framework;resolution:=optional;version:="↩
    1.3.0"
Service-Component: OSGI-INF/services.xml
Bundle-NativeCode: lib/mylib1.dll;lib/mylib2.dll;osname=Win32;↩
    processor=x86,...
Require-Bundle: javax.activation;resolution:=optional,javax.mail;↩
    resolution:=optional
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
```

Manifest file misses one of the most important information in an OSGi application – usage of services. Service is the only communication channel in an OSGi framework, even events are realized through services. It is an interface, that is registered by a bundle to inform the others that it is able to perform actions specified by this interface. Other bundles can ask OSGi framework to handle them this interface so they can use it. All services are registered or obtained directly in the source code of every bundle – thus making it hard to gather the information. Registering services in code is plain OSGi approach however there are technologies that enables easier registration of services. These technologies are called "‘Declarative Services"’ and "‘Spring Dynamic Modules"’, both will be discussed later on. This OSGi plug-in is able to handle all three approaches of service registration.

This is the reason to use bundles in binary form, because loader can take advantage of the ASM library[5] used for bytecode manipulation and analysis. Thanks to this library it is easier to get the list of all called methods and identify calls that are used for manipulation with services. The methods capable of making these manipulations are `registerService()`, `getServiceReference()`, `getServiceReferences()` which cover all possible ways of providing or requiring service. These methods are therefore found using ASM and added to the description of each bundle.

Moreover there are two additional technologies that helps developers to register and acquire services on one place. The first of them is called “Declarative Services” and it is part of the latest OSGi release (4.2). It defines these information in special XML file with predefined format. The second

[5]http://asm.ow2.org/

is called “Spring DM”[6] and have a lot of common with the ”‘Declarative Services”’. It also uses XML file to define connections between services, but instead of declarative services it uses a well known Spring notation to define the same. The OSGi loader is able to read both these XML files and retrieve the necessary information from them.

Finally when all bundles were analyzed it is time to check relations. There are five types of relations we identified for an OSGi applications, bundles can be have these relations – exported or imported package, required bundle and provided or required service. All these attributes are used to create expected connections and thus provide the structure of this OSGi application.

7.4.2 EJB 3

EJB loader requires a compiled JAR archive with complete EJB application as an input. WAR and EAR archives are not supported directly, however they contain compiled JAR anyway. Almost all the information about beans (EJB components) is directly in the source code in form of Java annotations. Every class in the provided archive has to be analyzed in order to decide if it contains some important information. However some additional information might be found in deployment descriptor XML file stored in `META-INF/ejb-jar.xml` which has to be also analyzed. ASM library is used for analysis of Java classes, which proved to provide a rich palette of features earlier in an OSGi loader.

EJB is the only component model analyzed by loaders that defines several types of components. The most common and basic one is called “Session-Bean” annotated with `@Stateful` or `@Stateless` defining the type of session bean. Stateless session beans are intended to perform individual operations automatically and do not maintain state across method invocations. The data maintained by stateful session beans are intended to be transitional. It is used solely for a particular session with a particular client. An example of a stateful session bean is in Listing 7.5. Session beans can be moreover annotated with `@Remote` and `@Local` defining if this bean is accessible only locally or also by remote calls.

Listing 7.5: A stateful session bean implementing a remote interface

```java
@Stateful(name="parkoviste")
@Remote({ParkovicteRemote.class})
public class ParkovisteBean implements ParkovisteRemote {
  ...
}
```

[6]http://www.springsource.org/osgi

Second type of component is called "MessageDrivenBean" annotated with `@MessageDriven`. All information required to create a proper link to both event types (topic or queue) are defined directly in the annotation as shown in Listing 7.6. The most important parameter is `mappedName` which define the JNDI name of topic/queue and the rest of parameters is in `activationConfig`.

Listing 7.6: A message driven bean complete annotation

```
@MessageDriven(mappedName="jms/ParkovisteEvents", activationConfig={
    @ActivationConfigProperty(
      propertyName="acknowledgeMode", propertyValue="Auto-↩
          acknowledge"),
    @ActivationConfigProperty(
      propertyName="destinationType", propertyValue="javax.jms.Topic↩
          "),
    @ActivationConfigProperty(
      propertyName="subscriptionDurability", propertyValue="Durable"↩
          ),
    @ActivationConfigProperty(
      propertyName="clientId", propertyValue="PultBean"),
    @ActivationConfigProperty(
      propertyName="subscriptionName", propertyValue="PultBean")
})
public class PultEventBean implements MessageListener { ... }
}
```

Session and message driven beans have some features common. They can use a powerful injections making an easier initialization of properties. Annotation `@Resource` can be used to inject almost any object easily, without any complex retrieval method. When resource annotation contains a parameter name it is used to connect this resource with resource defined in deployment descriptor XML. One can use it to get event topics or queues, global variables and other beans. Yet another annotation makes it easier to create and use web services.

The last component type is called "Entity" annotated with `@Entity`. This bean is just simple DAO (Data Access Object) mapped directly on table in database. It can set a lot of features directly by using annotation, specifying the type of expected data – making it more resilient to errors. An example of such mapping is in Listing 7.7. Entities does not have any additional features.

Listing 7.7: An entity bean mapped on database

```
@Entity
@Table(name="CATEGORIES")
public class Category implements Comparable<Category>, Serializable ↩
    {
  @Id
  @Column(name="ID")
  private long identifier;
  @Column(name="NAME", length=25, unique=true, nullable=false)
  private String name;

  @OneToMany
  @Transient
  private Collection<Product> products;

  ...
}
```

Relations between all these types of components are created at the end, when all classes are analyzed. EJB loader generally recognizes the relations between provided and required interfaces, event publishers and listeners and web service definitions and references. Loader also recognizes the usage of entity in other types of beans and creates connection. All these relations are analyzed by static analysis, therefore they might not correspond to the relations made by the application in run-time. The loader ends when all these relations are created.

7.4.3 SOFA 2

SOFA 2 loader is able to work with two different types of sources – 1) remote SOFA repository and 2) local project directory. In both cases all information required by loader are stored in special SOFA definition files called ADL, with structure based on type of source. SOFA is a hierarchical component model thus components can be created from other components. SOFA recognizes several types of building blocks and “Assembly” is the top entity. A simple assembly is in Listing 7.8. It uses another entity “Architecture” as a building blocks, architecture stands for basic component in SOFA. Every assembly defines a top level architecture and the list of subcomponents. The top level architecture defines how does the whole application look, in other words it is the main component.

Architecture is composed of the implementation and definition part. Definition part is called “Frame” and it is the lowest level entity. Implementation is just a class that implements what was defined in frame. Any architecture can have a list of subcomponents. Frames define which interfaces are provided and required and which communication style is used. This structure is valid for both types of source, however the remote repository is more oriented on versioning which is already a part of all ADL files.

Listing 7.8: Assembly entity defined by local ADL file

```
<?xml version="1.0" encoding="UTF-8"?>
<assembly name="org.objectweb.dsrg.sofa.examples.logdemo.assm.↩
    LogDemo" top-level-arch="sofatype://org.objectweb.dsrg.sofa.↩
    examples.logdemo.arch.LogDemo">
  <subcomponent name="logger" arch="sofatype://org.objectweb.dsrg.↩
      sofa.examples.logdemo.arch.Logger" />
  <subcomponent name="tester" arch="sofatype://org.objectweb.dsrg.↩
      sofa.examples.logdemo.arch.Tester" />
</assembly>
```

SOFA might have a complex structure, however the loader is quite straightforward as it needs only to analyze a list of XML files. The user can select between two modes – first can load application from a repository and second will work with local files. The format of XML file varies based on the type of source. Loader uses SOFA libraries to connect to the remote repository and "Cushion" tool for local project directory searches. It uses one assembly as an input for the whole analysis, because assembly defines the whole application. Loader then creates the tree of dependencies and prepare all ADL files for future analysis.

At the end SOFA loader has to find all relations between all components. It recognize two types of relations – 1) hierarchical, which are already known and 2) non-hierarchical, which has to be analyzed. Non-hierarchical relations can be created only between components on the same level of hierarchy in the scope of subcomponents. When all these relations are discovered the analysis process ends.

Chapter 8

AIVA - The Implementation

The concept of AIVA was introduced in Chapter 6. Here we will provide some implementation details, that are relevant to its design and architecture, but the description will not go as low as an algorithmic level of details. Some techniques were already suggested in Chapter 6. These techniques require user-defined data, however the design of how are these data created was not discussed. These information are also part of this chapter.

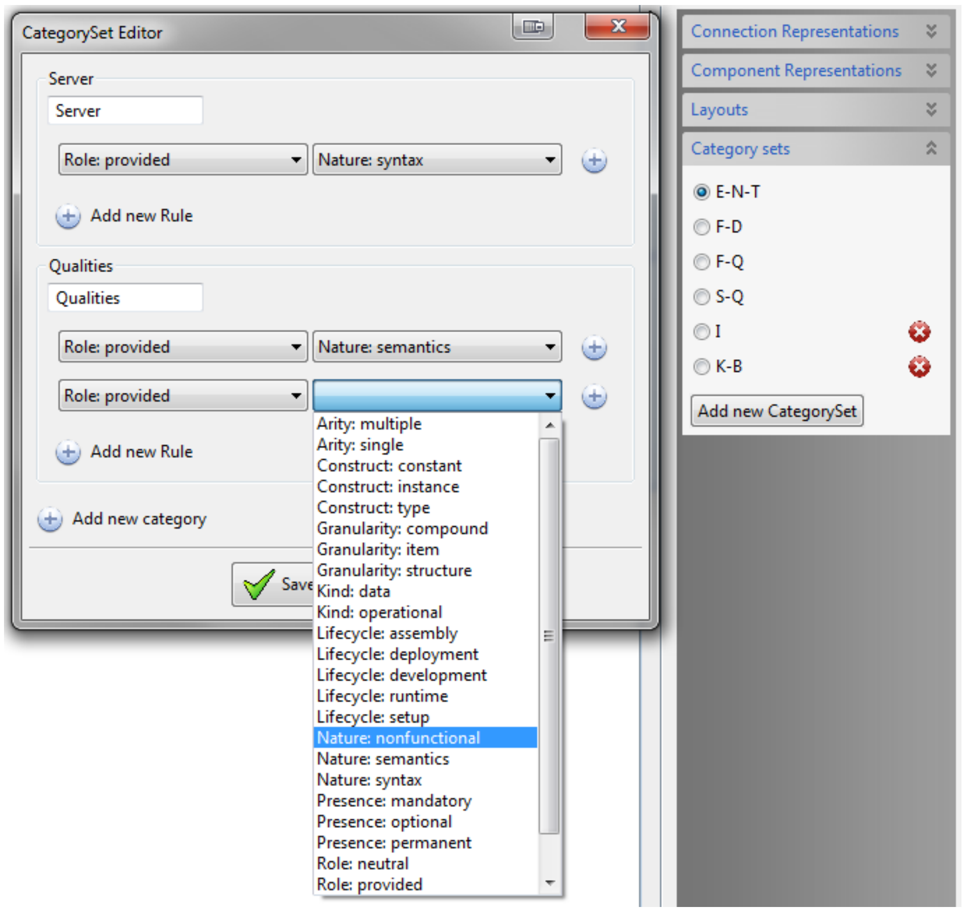
AIVA was implemented as a plug-in for the ComAV platform. It implements all features described earlier in Chapter 6 and can make advantage of features that are offered by the ComAV as mentioned in Chapter 7. Therefore this implementation does not have to develop its own user interface, create its own loaders of component-based applications or worry about mechanisms used to save or load user data. All of this is accomplished by design, as was mentioned earlier, so AIVA implementation is just one of visualization approaches that will use advantages of ComAV.

8.1 Visual Editors

This Section will describe visual editors that had to be designed so users might make use of several AIVA features. These features are namely Category Sets, described in Section 6.4, and conditional rules, described in Section 6.8. These visual editors are used to create user-defined data, that are used to personalize the view provided by AIVA.

8.1.1 Creating New Category Sets

It was already mentioned that new category sets can be added, but the way how to do it was not discussed yet. AIVA provides a powerful graphical


CategorySet Editor

Server

Server

Role: provided | Nature: syntax

Qualities

Qualities

Role: provided | Nature: semantics

Role: provided | Nature: nonfunctional

Figure 8.1: Editor used to create new category sets

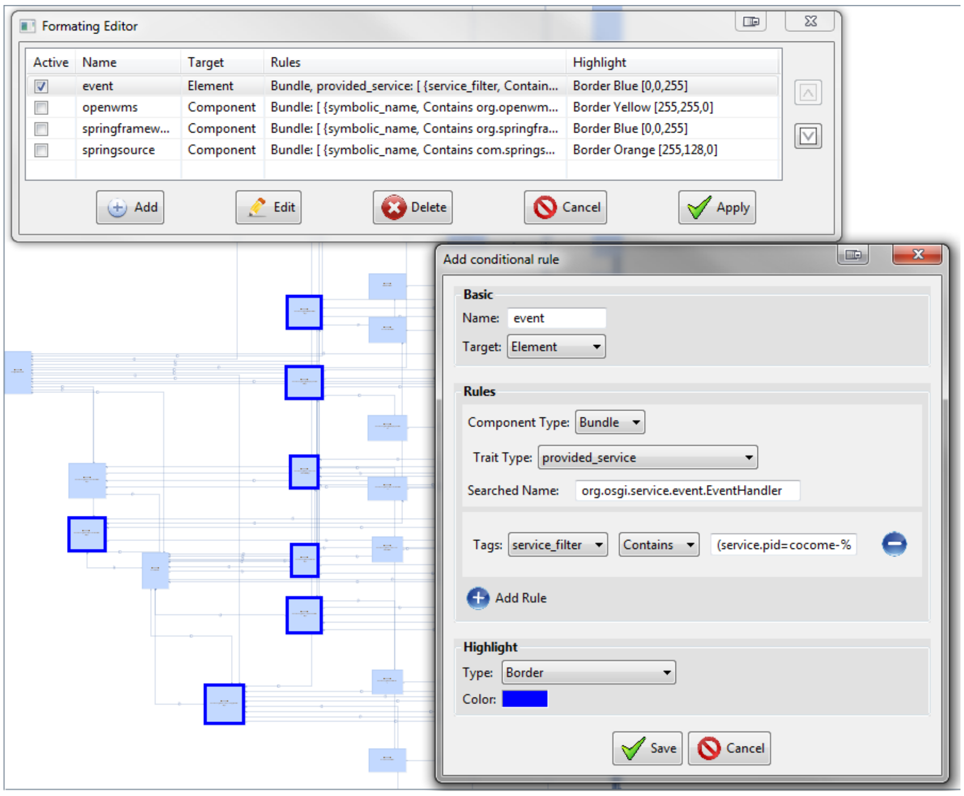
editor that enables simple creation of new category sets. Figure 8.1 shows how to create new sets on already discussed S-Q view. One can interactively add required number of categories, rule sets and rules. Every line of rules in this editor is called rule set and applies a logical operator AND between its rules. One category can define more than one rule set using a logical operator OR between these rule sets.

Created category set is then saved in a XML definition file for later use and new option appears in the AIVA menu, which is also in Figure 8.1. These sets are available for future use in all projects of active workspace and one can switch interactively between different ones.

8.1.2 Condition Editor

AIVA introduced quite a lot of ideas how to improve “classical” conditional highlighting in Section 6.8. Most of these ideas were focused on how to use advantages of the ENT meta-model in order to create more complex and advanced conditional rules. All rules are component model specific to offer better customization. Therefore the visual editor must be aware of component model and work only with rules applicable for it.

Editor of condition rules consists of two parts. Both of these parts are captured in Figure 8.2. First part works as a manager of defined rules, shown in the upper part of the Figure. Second part helps to define, or edit, new condition rules.


Formating Editor

Active	Name	Target	Rules	Highlight
✓	event	Element	Bundle, provided_service: [{service_filter, Contain...	Border Blue [0,0,255]
	openwms	Component	Bundle: [{symbolic_name, Contains org.openwm...	Border Yellow [255,255,0]
	springframew...	Component	Bundle: [{symbolic_name, Contains org.springfra...	Border Blue [0,0,255]
	springsource	Component	Bundle: [{symbolic_name, Contains com.springs...	Border Orange [255,128,0]

Add conditional rule

Basic
Name: event
Target: Element

Rules
Component Type: Bundle
Trait Type: provided_service
Searched Name: org.osgi.service.event.EventHandler
Tags: service_filter Contains (service.pid=cocome-%

Highlight
Type: Border
Color: 🟦

Figure 8.2: Editor used to create and manage different condition rules.

Management window is the main one. It is able to load and save defined condition rules on hard drive. These rules are saved as a XML definition files in a ComAV workspace. All condition rules applicable for the component model of visualized application are loaded and ready for use. Therefore condition rules defined previously to highlight information in any other application are available. These rules might not be relevant for the current application, but it is more user friendly than to force users to create the same rules again. Management window offer all standard functionality required to easily manage created conditions – create, edit, delete. It is able to simply select required conditions, which will all be applied. Finally it is able to order created conditions, because they are applied bottom to top, therefore condition on top will apply its highlighting as the last and it will remain visible.

Creation window is used to define new conditions. One has to fill a name of the rule and select if he wishes to create rules for components or its elements. Second step is to select type of component from list, when component model has more types; and type of trait if the condition is aimed on elements. Name of component/element is given then. It is possible to use wild card character (%) for any number of characters, therefore % symbol alone stands for any name. Finally it is possible to define any number of rules on tags specified for the selected component type, or trait type. Rules can check if the value is in the specified range or if it contains specified string. Wild card character is also enabled. When all rules are created it is still necessary to choose the style of highlighting.

AIVA enables visual customization of highlighting by selecting color and type. There are following types of highlighting: border (thick line around the component), panel background (the whole background of component), overview background (components are colored in the overview), element background (element background is colored) and element font (color of font is changed). Figure 8.2 shows creation of rule in OSGi application that helps to highlight components, that generate events with specified id.

8.2 Structure of the AIVA

The list of all plug-ins was already provided in Section 7.3 thus their names will be referenced. JGraphBasic was originally a part of AIVA, however as the work on AIVA continued a new need emerged. JGraphX[1] is an external graph library, that could be used in more than one visualization style. AIVA contained a lot of functionality that modified JgraphX library to be used in the ComAV environment. Even more functionality could be shared between different visualizations. These were the reason why these two plug-ins were separated. More general parts of AIVA were moved in JGraphBasic and the most specific ones were kept in AIVA plug-in.

The relation between JGraphBasic and AIVA is therefore very close, AIVA uses almost all functionality of JGraphBasic. Next to this plug-in it also requires ENTMM, ComAV libraries and ComAV core plug-ins.

[1]http://jgraph.com/jgraph.html

8.3 JGraphBasic Plug-in

JGraphBasic is an abstract plug-in to be used by various other visualization plug-ins. It encapsulates JgraphX library, it connects the ENT implementation with this library and it offers several utilities to be used by the visualization plug-ins.

JGraphBasic conforms to the specification of Visualization Extension Point, defined in Section 7.3.2. Visualization plug-ins need only to extend Jgraph-Basic, therefore the developers of these plug-ins does not need to understand to the RCP. JGraphBasic's main package is `cz.zcu.kiv.comav.visualizations.j-graphbasic`, with four direct subpackages. JGraphBasic consists of almost 50 different classes that are sorted according to their purpose.

The main subpackages that contain almost all the functionality of plug-in are:

- `cz.zcu.kiv.comav.visualizations.jgraphbasic.connection` handles how are connection lines drawn and how they look like, e.g., clean line, line with lollipop symbol.
- `cz.zcu.kiv.comav.visualizations.jgraphbasic.ent` is used to map the ENT structure into the graph nodes, defined by JGraph. The most important class is `CommonGraphModel` that describe mapping of all nodes and connections in graph. It also contains features, that support the ENT categories.
- `cz.zcu.kiv.comav.visualizations.jgraphbasic.overlay` provides a group of class-/es, that are used to create a rich component representation. JGraphX can basically use only simple html in its body, which is the reason why this overlay feature was created. It also includes all component representations defined in Section 6.3. These component representations can be used in other visualization styles other than AIVA.
- `cz.zcu.kiv.comav.visualizations.jgraphbasic.utils` contains various utilities that are used either by JGraphBasic itself or exported to be used by visualization plug-ins. These utilities are for example resource and message manager, different formatting tools and different object comparators.

8.3.1 Connecting the ENT Structure with JGraphX

One of the most important parts of JGraphBasic is how are the ENT structure and JGraphX connected. This relation between graphical and data part is essential to any visualization style, that could make use of JGraphBasic.

Therefore its implementation will be described in more detail than the rest of JGraphBasic plug-in. There are three classes from the ENTMM plug-in – Component, Element, Binding. They are sufficient to create a solid relation between data and visual part.

Listing 8.1: ComponentNode - connects the ENT component with the JGraphX cell

```
public abstract class ComponentNode {
    protected mxGraphComponent graphComponent;
    protected Component component;
    protected Object cell;

    /**
     * Returns label in a UML style
     */
    public String getLabel() {
        String t = "<i>\u226a" + component.getDef().getCtname() + "\↩
            u226b</i>";
        return "<center>" + t + "<br><b>" + component.getName() + "↩
            </b></center>";
    }

    ...
  // Constructor and other getters and setters
  ...
}
```

Listing 8.2: Connection - connect the ENT local and alien component with the JGraphX edge

```
public class Connection {
    private Component localComponent;
    private Component alienComponent;
    private Object edge;

    ...
  // Constructor and other getters and setters
  ...
}
```

Classes in Listings 8.1 and 8.2 show only a very simple objects used to map these different types of data. On the other hand, class in Listing 8.3 represent the main graph object. It contains all the necessary information and functions that helps to find required nodes, connections and elements based on different input.

Listing 8.3: CommonGraphModel - provide the list of all nodes and connections

```
public class CommonGraphModel {
    protected List<VisualComponentNode> nodes;
    protected List<Connection> connections;
    protected List<Component> componentList;

    /**
     * Returns connection, based on the JGraphX edge
     */
    public Connection getConnection(Object edge) {
      ...
    }

    /**
     * Returns Node, based on the ENT component
     */
    public VisualComponentNode findNodeToComponent(Component ↩
        component) {
      ...
    }

    /**
     * Returns Node, based on the JGraphX cell
     */
    public VisualComponentNode findNodeToCell(Object cell) {
      ...
    }

    /**
     * Returns the list of all elements that create connection ↩
        between two components.
     * Works only with the ENT structure.
     * @param localBindings {@link Binding} of local {@link ↩
        Component}
     * @param alienComponent alien {@link Component} in {@link ↩
        Connection}
     * @return {@link List} of trait {@link Element}s that are part ↩
        of binding from localBindings and alienComponent
     */
    public List<Element> getConnectedElements(Binding[] ↩
        localBindings, Component alienComponent) {
      ...
    }

    ...
  // Constructor and other getters and setters
  ...
}
```

Other interesting aspect is the use of CategorySets. The ENT implementation itself does not support them, it provides only the classification – because ENTMM was developed using MDD (Model Driven Development) from a UML diagram. Thus, to keep this MDD clean we decided to move CategorySets to the JGraphBasic.

JGraphBasic defines class CategorySet, in Listing 8.4. It is used to describe one CategorySet – its name and its categories. It is also used to sort traits of one component, into categories of the CategorySet, using category rules and element classification.

Listing 8.4: CategorySet - filter and group traits based on the defined categories

```java
public class CategorySet {
    private String fullName;
    private String shortName;
    private ArrayList<Category> categories;

    /**
     * This method will sort all of the traits in the corresponding
     * categories and will return all the information for further ↩
         processing.
     *
     * @param component − component to be processed
     * @return traits in categories
     */
    public TreeMap<Category, ArrayList<Trait>> getTraitsInCategories↩
        (Component component) {
        ...
    }

    ...
  // Constructor and other getters and setters
  ...
}
```

8.3.2 Node overlays

JGraphX basically support only a simple HTML in a body of nodes, however it is possible to set overlay component with the same size. These overlays are directly supported by JGraphX library. The only requirement is, that this overlay has to extend `JComponent` class – the elementary Swing graphical unit.

Listing 8.5 shows an abstract class, that should be used for these purposes. It has reference on all objects, required for visualization of the ENT component – component, categorySet and cell in a graph. It defines a lot of abstract methods, that can be used for different kinds of highlighting and formatting – as defined in AIVA. Other visualization styles does not have to use all these features or can add different ones. The most simple extension of `BasicCellOverlay` will hide JgraphX cells in the graph with empty Swing components.

Listing 8.5: BasicCellOverlay - provide a basic graphical component to represent components

```java
public abstract class BasicCellOverlay extends JComponent implements↩
     ICellOverlay {
    protected Component component;
    protected CategorySet categorySet;
    protected mxGraphComponent graphComponent;
    protected Object cell;
    ...
  // Constructor and other getters and setters
  ...
  // A lot of abstract methods -- highlighting and formatting
  ...
}
```

The example usage of overlay is provided in Listing 8.6 that shows the implementation of an AIVA tree view. It does not have to bother with the details on how to create JGraphX overlay, it merely extend `BasicCellOverlay` and create the look of component. New graphical elements can be added on the empty component in the `initialize()` method. All abstract methods should be implemented in order to use this overlay in the AIVA.

Listing 8.6: ComponentTreeOverlay - implementation of an AIVA tree view

```
public class ComponentTreeOverlay extends BasicCellOverlay {
    public static final String NAME = Messages.↩
        COMPONENT_TREE_REPRESENTATION;
    protected JTree tree;
    protected JScrollPane scroller;
    // More GUI elements
    ...

    /**
     * This method is related to jGraph. It adds detailed ↩
         information to a
     * simple cell.
     */
    protected void initialize() {
        // create content of vertex
        setLayout(new BorderLayout(10, 10));

        ...
    }

    ...
  // Constructor and other getters and setters
  ...
  // Implementation of abstract methods -- highlighting and ↩
     formatting
  ...
}
```

8.4 AIVA Plug-in

AIVA plug-in uses JGraphBasic and extends it to provide visualization as defined in Section 6. It uses basic features provided by JGraphBasic, connects and complements them. AIVA consists of six direct subpackages and almost 60 classes.

The main package contains more than necessary plug-in classes. `Editor` and `AIVAGraph` are also put in the top hierarchy as these are the most important ones. `Editor` creates the whole AIVA workspace – all menus, settings and switchers. `AIVAGraph` is a graphical component, that draw the AIVA diagram itself.

The main subpackages, that provide AIVA's functionality are:

- `cz.zcu.kiv.comav.visualizations.aiva.connection` is used to add another connection style. AIVA defined an element oriented connection in Section 6.5.2. Subpackage `extendedlollipop` contains implementation of this functionality.
- `cz.zcu.kiv.comav.visualizations.aiva.handlers` contains descriptors that enable AIVA to be extended through RCP.
- `cz.zcu.kiv.comav.visualizations.aiva.hierarchy` implements support for composite component models, which was described in Section 6.6. It uses another instances of `AIVAGraph` to draw the structure of subcomponents. Overall integrity and functionality is addressed.
- `cz.zcu.kiv.comav.visualizations.aiva.highlighting` manages all the different types of highlighting described in the specification of the AIVA.
- `cz.zcu.kiv.comav.visualizations.aiva.utils` contain various types of utilities. Several managers that modify look and resources, action listeners that are used to provide described interaction techniques, or visual components like diagram overview.
- `cz.zcu.kiv.comav.visualizations.aiva.views` is used to add RCP views that contain additional information useful for visualization. AIVA uses only *Active Rules View* to show all rules activated in *Conditional Manager* 8.1.2.

8.4.1 Action listeners

Implementation of action listeners, mainly mouse/mousewheel listeners was a bit complicated, because these events had to be redirected between two frameworks – SWT (used by RCP, ComAV) and AWT/Swing (used by JGraphX). This brought several problems, however most of them were solved. The only problem that remains unfixed is touch control. More than one solution was tested but without satisfying results.

Listing 8.7: Editor - create the whole AIVA workspace

```
public class Editor extends EditorPart {
    public static final String ID = "cz.zcu.kiv.comav.visualizations↩
        .aiva.editor";
    ...
    // Other attributes and methods
    ...

    /**
     * This method creates all main components and places them in ↩
         the editor area
     */
    @Override
    public void createPartControl(Composite parent) {

      ...

        // Graph component and SWT listener
        final mxGraphComponent graphComponent = graph.↩
            getGraphComponent();
        SwtEventListener swtlistener = new SwtEventListener(↩
            graphComponent, swtAwtGraphComponent);
        swtAwtGraphComponent.addMouseWheelListener(swtlistener);

        ...

        // Add SwingListener
        AivaGraphMouseListener graphMouseListener = new
            AivaGraphMouseListener(graph, swtAwtGraphComponent, ↩
                getEditorSite().getShell());
        if (graphMouseListener != null)
            graph.getGraphComponent().getGraphControl().↩
                addMouseListener(graphMouseListener);
    }

}
```

Listing 8.7 shows that two different listeners have to be registered in order to manage both mouse and mousewheel events. Both these listeners require the reference on Swing graph component and on its SWT container. These references are required to ensure the stability of the application – to synchronize threads, when working in two different graphical frameworks.

The shorter listener is used to manage mousewheel event and is listed in Listing 8.8. Most of the code in method `mouseScrolled()` is used for synchronization purposes.

Listing 8.8: SwtEventListener - manages SWT mouse wheel event

```
public class SwtEventListener implements MouseWheelListener {

    private mxGraphComponent graphComponent;
    private Composite composite;

    public SwtEventListener(mxGraphComponent graphComponent, ↩
        Composite composite) {
```

```
        this.graphComponent = graphComponent;
        this.composite = composite;
    }

    @Override
    public void mouseScrolled(MouseEvent e) {
        final int count = e.count;

        EventQueue.invokeLater(new Runnable() {
            public void run() {
                if (composite.isDisposed() == false && composite.↩
                    getDisplay().isDisposed() == false) {
                    composite.getDisplay().syncExec(new Runnable() {
                        public void run() {
                            if (count < 0) {
                                graphComponent.zoomIn();
                            } else {
                                graphComponent.zoomOut();
                            }
                        }
                    });
                }
            }
        });
    }
}
```

Chapter 9

Evaluation of the Proposed Approach

The goal of this Chapter is to evaluate our new visualization approach called AIVA, which was described in Chapters 6 and 8. As a new approach it should be evaluated if the claimed advantages of this approach are true. The strongest and most important claims are that AIVA solves problems of overly complex component diagrams (Research challenges in Section 6.1) and that it should boost the learning process using interaction (defined as one of the goals of this thesis in Introduction). Both these claims were mostly defined against UML, thus comparison evaluation should prove which approach provides better results.

Complexity of diagrams was studied in a case study, that had clearly defined research questions and evaluation criteria. It compared diagrams made in a UML tool and in AIVA from several points of view, concluding which one of them provides a less complex diagram. This case study is thoroughly described in Section 9.2. This case study was already reviewed and published in one of our papers [60] and the content of Section 9.2 is almost identical.

The impact of interactivity on learning process is hard to measure. Therefore, we simplified our question and asked if interactivity will help users to find their answers faster. Such simplification lead to the implication that if user can find his answers faster he should also learn faster. A user study that monitored performance of several participants would evaluate such question. This study measured user performance in a set of several key tasks, analyzed results and provided discussion about findigs. The tasks measured were the elementary ones that are used the most frequently – more complex tasks are composed of them. All details about this user study are provided in Section 9.3 for the sake of replicability. This user study was also already reviewed and published in one of our papers [61] and the content of Section 9.3 is almost identical.

9.1 Overview of Used Tools and Technologies

This Section covers the description of tools and technologies that were used in the following studies.

9.1.1 RSA

IBM RSA (Rational Software Architect) was chosen as an example of an advanced UML tool, that fully supports UML 2.0. It offers all standard features like scrolling, zooming and outline view for basic navigation. It supports some advanced features that allow users to manipulate with the diagram like changing the layout of nodes, changing the line routing and modifying the look of components and interfaces.

RSA can scroll on component selected in outline – which helps to find it fast. When a connection line is selected, RSA centers the view on this line and slightly highlights it – it uses small rectangles in corners, but no color change and no highlight of connected components.

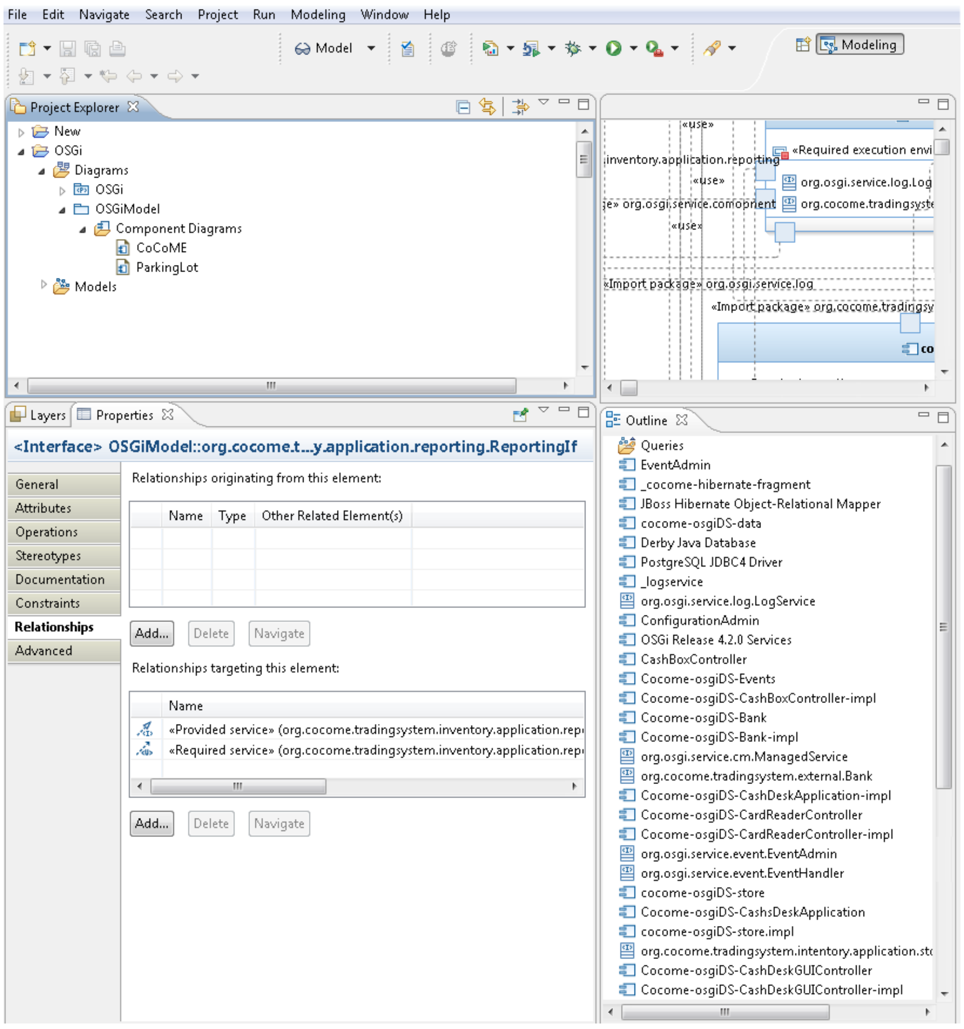
Added value is in its "*properties view*", displayed at the bottom of the screen. This view shows all the details about components and relations and, most importantly, it can be used to navigate to related components. For example, the "*Relationships*" tab shows a list of all elements that use or are used by the component. This list clearly specifies which kind of relation is used and which component is related. The name of the related component has the form of a link, so the user can easily find more information about it.

9.1.2 Technological Background

The comparison of UML and AIVA was performed on two applications. Both are implemented in OSGi which is a standard industry component model used for example to build Eclipse IDE. In this section we will briefly describe both the OSGi component model and the structure of these applications. UML models of these application are available later in the text as it was part of a case study.

OSGi Component Model – UML Profile

OSGi component model was referenced throughout the whole thesis. It was also used in our studies as both applications are developed in OSGi. AIVA



New
OSGi
Diagrams
OSGi
OSGiModel
Component Diagrams
CoCoME
ParkingLot
Models

«use»
inventory.application.reporting
«Required execution envi
«use»
org.osgi.service.log.Log
ie» org.osgi.service.component
org.cocome.tradingsyste
«use»
«Import package» org.osgi.service.log
«Import package» org.cocome.tradingsy

<Interface> OSGiModel::org.cocome.t...y.application.reporting.ReportingIf

Relationships originating from this element:

Name	Type	Other Related Element(s)

Relationships targeting this element:

Name
«Provided service» (org.cocome.tradingsystem.inventory.application.rep
«Required service» (org.cocome.tradingsystem.inventory.application.rep

Queries
EventAdmin
_cocome-hibernate-fragment
JBoss Hibernate Object-Relational Mapper
cocome-osgiDS-data
Derby Java Database
PostgreSQL JDBC4 Driver
_logservice
org.osgi.service.log.LogService
ConfigurationAdmin
OSGi Release 4.2.0 Services
CashBoxController
Cocome-osgiDS-Events
Cocome-osgiDS-CashBoxController-impl
Cocome-osgiDS-Bank
Cocome-osgiDS-Bank-impl
org.osgi.service.cm.ManagedService
org.cocome.tradingsystem.external.Bank
Cocome-osgiDS-CashDeskApplication-impl
Cocome-osgiDS-CardReaderController
Cocome-osgiDS-CardReaderController-impl
org.osgi.service.event.EventAdmin
org.osgi.service.event.EventHandler
cocome-osgiDS-store
Cocome-osgiDS-CashsDeskApplication
cocome-osgiDS-store.impl
org.cocome.tradingsystem.intentory.application.st
Cocome-osgiDS-CashDeskGUIController
Cocome-osgiDS-CashDeskGUIController-impl

Figure 9.1: Workspace of RSA

uses the ENT meta-model to define the specifics of concrete component model, such specification for the OSGi is in Section B. UML can use profiles to define the same – in a different way.

After a thorough study of the ENT OSGi model we developed a UML profile, so it can model the same information as the ENT meta-model. OSGi is a straightforward component model and thus it was possible to model all the information in a plain UML profile. However, it should be noted that UML meta-model is unable to describe the character of these information, thus it loses information. But this characteristics is not vital for the visualization itself. This UML profile of OSGi component model is presented in Figure 9.2.

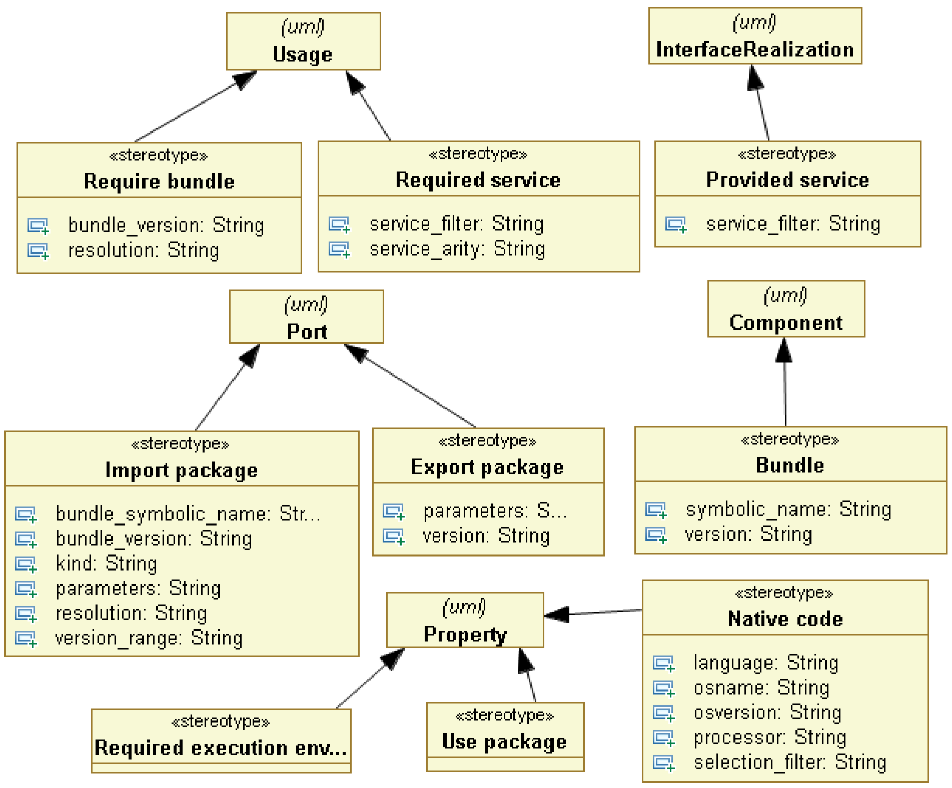


Figure 9.2: UML profile of OSGi component model

CoCoME

CoCoME stands for Common Component Modeling Example [49]. It is an example description of requirements specification captured in a form which would an supplier get from a business company. It was mainly developed for the purposes of comparing different approaches to a component based software development. It has been officially implemented by 13 modeling approaches; we implemented it in OSGi component model.

The described system serves as an information system for supermarket chains. It consist from three main parts. First is Cashdesk part, which contains the cash desk as it can be seen in supermarkets, including bar-code scanners, credit card readers etc. Second part is store infrastructure consisting from store server and store client. Finally there is an enterprise server, which consist from an enterprise server and client applications. CoCoME is assembled from 37 components and 12 interfaces, thus representing a medium-size application.

ParkingLot

ParkingLot is a simple example application, developed to demonstrate the features of CoSi component model[1]. It is assembled from 5 components with 4 interfaces. EventAdmin component had to be added to enable event handling, in an OSGi implementation, thus the whole application was extended on 6 components and 6 interfaces.

ParkingLot was chosen as a case example because it is small enough to be compared thoroughly in reasonable time.

9.2 Case Study on Structure Readability

This Section covers the first step in evaluation of AIVA. It will be thoroughly compared with UML component diagram extended with profile, which is currently the state of the art in visualization of component-based software. These two approaches are going to be compared on a case study of two applications: first is an application called ParkingLot and second is called CoCoME [23], which is a recognized component modeling example. Details about these examples are covered in Section 9.1.

Evaluation of a new approach by comparing with an existing one is recognized as a valid evaluation method classified in [28] and emphasized in [19]. We would like to note, that this is only a first step in our evaluation process.

9.2.1 Case Study Approach

This case study was inspired by a formal definition of a software visualization by von Mayrhauser [68]:

> Software visualization is a discipline that makes use of various forms of imagery to provide insight and understanding and to reduce the complexity of the existing software system under consideration.

Three research questions were formulated based on this definition. All of them are focused to determine on which approach is able to provide better insight and understanding and to reduce the complexity of diagram. The research questions asked in the study therefore were:

1. Which approach provides more logical data representation.

[1]http://www.assembla.com/spaces/cosi/wiki/Tutorial_examples

2. Which approach provides less complex diagram.
3. Which approach provides better readable structure/hierarchy.

Evaluation Criteria

To correctly answer these research questions we defined several evaluation criteria, as follows.

To decide which approach provides *better data representation* we used these criteria: 1) How are these data organized logically; 2) What can be learned on first look; 3) How accessible are secondary information (attributes of stereotype described in Figure 9.2). All these criteria are focused on how easy it is to get and classify an information.

The *complexity of diagram* can be quantified, because diagram with less graphical elements is less complex. There are two major types of graphical elements, that crate diagram more complex: 1) Number of nodes; 2) Number of lines.

The *readability of structure/hierarchy of diagram* is individual and depends on the structure of a visualized graph. The criteria to answer this question are: 1) How well is hierarchy readable from spatial organization of nodes; 2) How logically are nodes arranged.

Preparation of data

AIVA is able to reverse-engineer OSGi applications and thus both applications were automatically analyzed by it. AIVA then used this data to create oriented graph and applied its hierarchical layout on it.

The RSA was chosen as a UML tool, because of its importance and richness of features. Both UML diagrams were created manually, because there is no OSGi to UML reverse engineering tool. These diagrams are based on information from AIVA diagrams to ensure that it present the same structure and data. There are a few components, that are not complete in UML as they contain dozens of elements that are not related to the application itself as they are imported by OSGi framework. Hierarchic algorithm was used on CoCoME diagram without any manual modification.

The diagrams of CoCoME are too big to be a part of this paper in readable resolution so we created a preview of these diagrams to illustrate our statement. Full size images can be viewed on project homepage[2] together with all resources needed to recreate our experiment.

[2]http://www.assembla.com/spaces/comav/wiki/Comparison_of_AIVA

9.2.2 Results of the case study

The case study data were gathered by visualizing the two applications in both visualization approaches and comparing the final qualities of the diagrams.

To answer the first research question about quality of data representation, it is better to use the case example of ParkingLot, which is representant of small applications. All criteria concerned with this question will be answered in one paragraph in following subsections. ParkingLot is visualized in Figures 9.3 and 9.4. Larger version of these figures can be found in Appendix C.

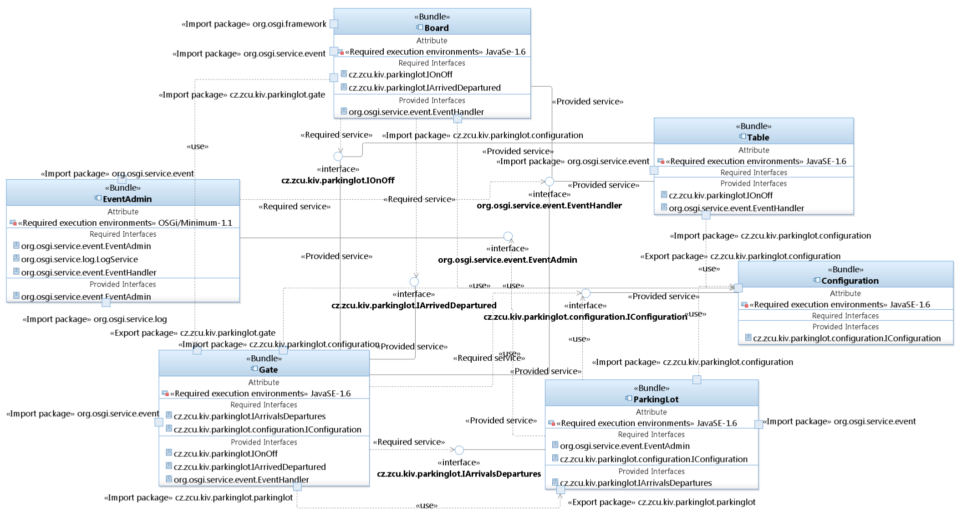

«Bundle» Board
«Required execution environments» JavaSE-1.6
cz.zcu.kiv.parkinglot.IOnOff
cz.zcu.kiv.parkinglot.IArrivedDepartured
org.osgi.service.event.EventHandler
«Bundle» Table
«Bundle» EventAdmin
«Required execution environments» OSGi/Minimum-1.1
org.osgi.service.event.EventAdmin
org.osgi.service.log.LogService
«Bundle» Configuration
cz.zcu.kiv.parkinglot.configuration.IConfiguration
«Bundle» Gate
cz.zcu.kiv.parkinglot.IArrivalsDepartures
«Bundle» ParkingLot
«Import package» org.osgi.framework
«Import package» org.osgi.service.event
«Import package» cz.zcu.kiv.parkinglot.gate
«Import package» cz.zcu.kiv.parkinglot.configuration
«Export package» cz.zcu.kiv.parkinglot.configuration
«Import package» cz.zcu.kiv.parkinglot.parkinglot
«Export package» cz.zcu.kiv.parkinglot.parkinglot
«Required service»
«Provided service»
«use»
«interface»

Figure 9.3: ParkingLot in UML

UML

Regarding data representation, UML does not provide any logical organization of its provided/required elements. Both types of packages are mixed together on the edges of component as ports and user has to study which stereotype was applied on the port to decide which type it is. UML tools can change the appearance of components – to show/hide the list of elements in the body of component like interfaces in Figure 9.3. This feature makes it easier to get an overview of component, but UML supports only the most basic grouping and shows only some types of features in these lists.

Regarding diagram readability, in small diagram one can see almost any information at first sight. The packages are nicely showed as ports, so it is possible to recognize the type of relation – service/package/bundle. The

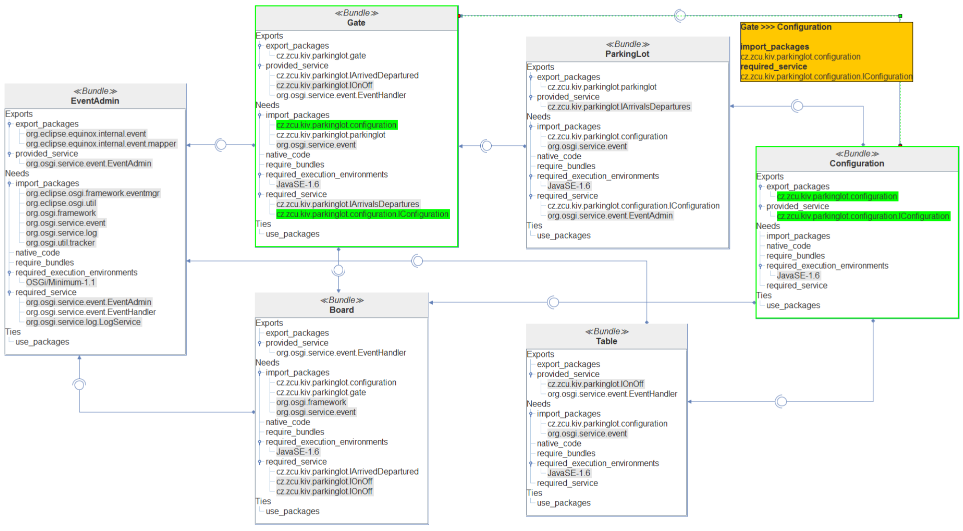

≪Bundle≫
Gate
Exports
export_packages
cz.zcu.kiv.parkinglot.gate
provided_service
cz.zcu.kiv.parkinglot.IArrivedDepartured
cz.zcu.kiv.parkinglot.IOnOff
org.osgi.service.event.EventHandler
Needs
import_packages
cz.zcu.kiv.parkinglot.configuration
cz.zcu.kiv.parkinglot.parkinglot
org.osgi.service.event
native_code
require_bundles
required_execution_environments
JavaSE-1.6
required_service
cz.zcu.kiv.parkinglot.IArrivalsDepartures
cz.zcu.kiv.parkinglot.configuration.IConfiguration
Ties
use_packages
Gate >>> Configuration
import_packages
cz.zcu.kiv.parkinglot.configuration
required_service
cz.zcu.kiv.parkinglot.configuration.IConfiguration
≪Bundle≫
ParkingLot
Exports
export_packages
cz.zcu.kiv.parkinglot.parkinglot
provided_service
cz.zcu.kiv.parkinglot.IArrivalsDepartures
Needs
import_packages
cz.zcu.kiv.parkinglot.configuration
org.osgi.service.event
native_code
require_bundles
required_execution_environments
JavaSE-1.6
required_service
cz.zcu.kiv.parkinglot.configuration.IConfiguration
org.osgi.service.event.EventAdmin
Ties
use_packages
≪Bundle≫
EventAdmin
Exports
export_packages
org.eclipse.equinox.internal.event
org.eclipse.equinox.internal.event.mapper
provided_service
org.osgi.service.event.EventAdmin
Needs
import_packages
org.eclipse.osgi.framework.eventmgr
org.eclipse.osgi.util
org.osgi.framework
org.osgi.service.event
org.osgi.service.log
org.osgi.util.tracker
native_code
require_bundles
required_execution_environments
OSGi/Minimum-1.1
required_service
org.osgi.service.event.EventAdmin
org.osgi.service.event.EventHandler
org.osgi.service.log.LogService
Ties
use_packages
≪Bundle≫
Configuration
Exports
export_packages
cz.zcu.kiv.parkinglot.configuration
provided_service
cz.zcu.kiv.parkinglot.configuration.IConfiguration
Needs
import_packages
native_code
require_bundles
required_execution_environments
JavaSE-1.6
required_service
Ties
use_packages
≪Bundle≫
Board
Exports
export_packages
provided_service
org.osgi.service.event.EventHandler
Needs
import_packages
cz.zcu.kiv.parkinglot.configuration
cz.zcu.kiv.parkinglot.gate
org.osgi.framework
org.osgi.service.event
native_code
require_bundles
required_execution_environments
JavaSE-1.6
required_service
cz.zcu.kiv.parkinglot.IArrivedDepartured
cz.zcu.kiv.parkinglot.IOnOff
cz.zcu.kiv.parkinglot.IOnOff
Ties
use_packages
≪Bundle≫
Table
Exports
export_packages
provided_service
cz.zcu.kiv.parkinglot.IOnOff
org.osgi.service.event.EventHandler
Needs
import_packages
cz.zcu.kiv.parkinglot.configuration
org.osgi.service.event
native_code
require_bundles
required_execution_environments
JavaSE-1.6
required_service
Ties
use_packages

Figure 9.4: ParkingLot in AIVA, with one connection (Gate – Configuration) selected

UML is basically supposed to provide all information at first sight. It comes handy that one can get the same experience from printed diagram and by using a big format/resolution printer to get even better context of the whole application. However, more complex diagrams make it harder to read any of these information.

Concerning the information which is not directly visualized but can be accessed manually, their availability depends on UML tool used. RSA needs four steps to get to these information: 1) select Properties view, 2) choose Stereotypes tab, 3) find Stereotype properties list, 4) locate relevant properties (RSA shows all inherited and empty properties). Other UML tools do not provide much better experience.

AIVA

Regarding data representation, AIVA provides grouping of elements based on their characteristics and lists them all on one place. The tree hierarchy is used in Figure 9.4 to emphasize where every element belongs. One can though easily classify each element. It is possible to change the grouping of elements on the fly – like when one needs to study what elements are functional and which are only the data. It is also possible to change representation of components, AIVA provides three completely different representations that are focused on logical arrangement of its elements and use easily comprehensible lists in its body.

Regarding diagram readability, it is possible to see all elements of a

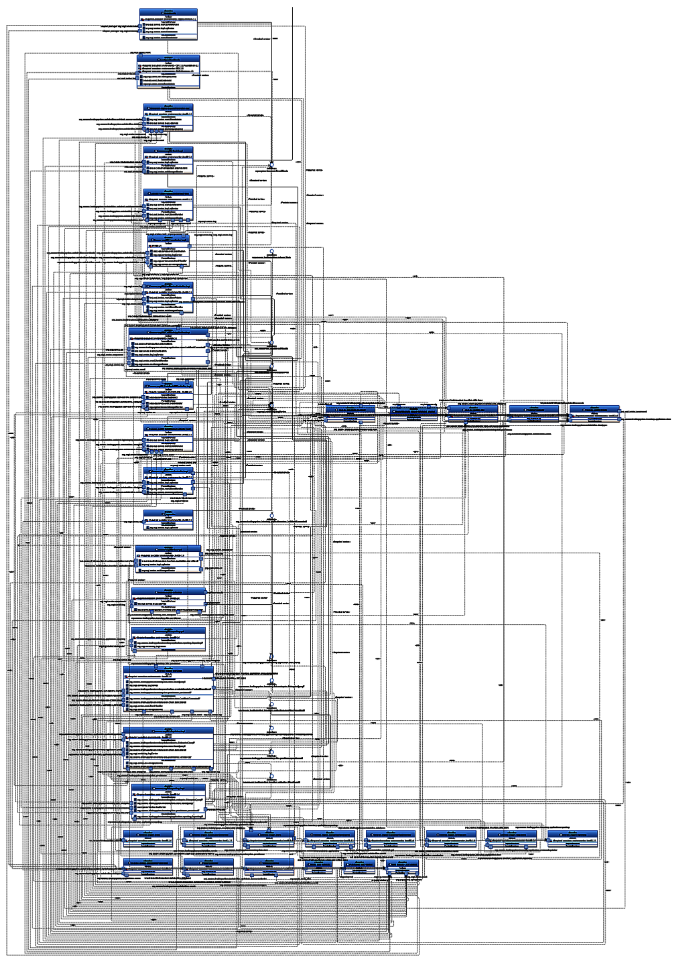


Figure 9.5: Complete CoCoME visualized with UML

component in one place as well as relations between components. AIVA collapses all lines between two components into one line to lower the complexity of diagram and emphasize components themselves. It does not provide any textual hint that would tell more about these relations – interaction is required. AIVA also uses only one visual style for connection line (same for service, package, bundle) and does not use any other graphical indicators like ports. AIVA is therefore cleaner and its readability is better in more complex diagrams, but requires more user interaction to get the same information which UML provides directly.

This information is not directly visualized and interaction is required. A tool-tip with the list of nonempty properties appears after hovering over the element, when no tool-tip shows after interaction it means that this element does not offer any additional information.

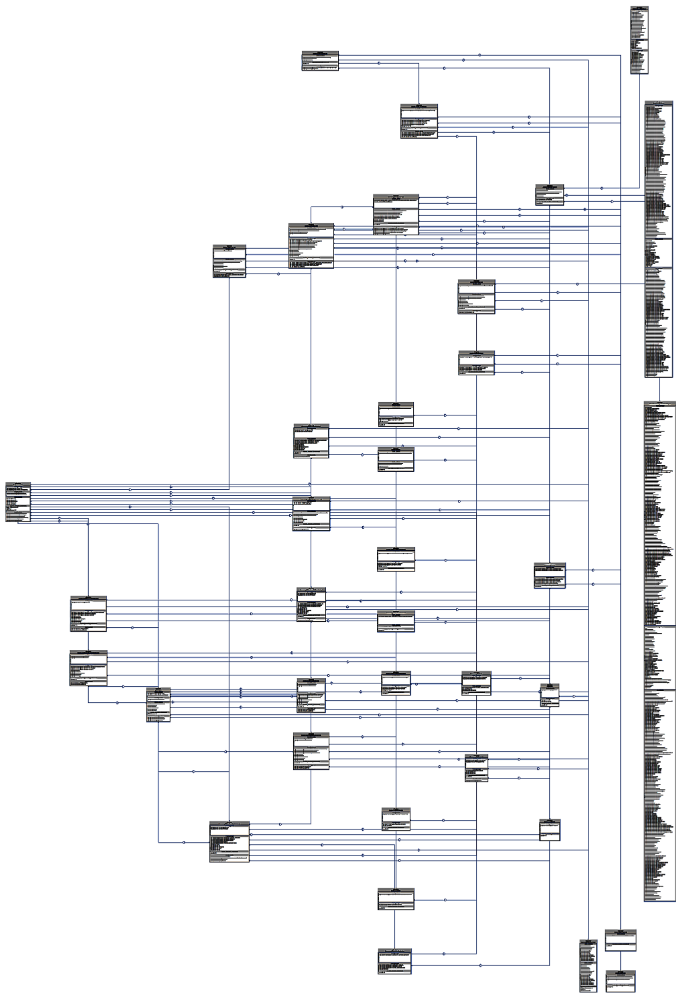


Figure 9.6: Complete CoCoME visualized with AIVA

Quantitative evaluation

With respect to diagram complexity measures, one can look on Figures 9.3 and 9.4, which are both well readable, and count the connection lines in the diagrams. AIVA uses 6 nodes and 11 lines, while UML has to use 12 nodes (6 components and 6 interfaces) and 20 lines to express the same. The hierarchy layout of components in ParkingLot is better distinguishable from AIVA diagram, but this statement is arguable on application of this size. ParkingLot diagrams can't answer this question, but they create a good basis for the next example.

CoCoME is visualized in Figures 9.5 and 9.6. AIVA uses 37 nodes and 125 lines, while UML has to use 49 nodes and 244 lines. Both diagrams contain the same amount of information, however UML needs two times more graphical elements to express the same. The structure of CoCoME is well readable from AIVA, one can immediately study the hierarchy layout of components and find the key components of the application, however UML diagram is flat – no spatial information can be observed from the diagram and the positioning does not seem to have any logical reason.

Summary of findings

This subsection presents answers to research questions.

1. **Which approach provides more logical data representation.** AIVA is better in two criteria of the first research question – it provides better logical organization of data and secondary information is better accessible. However it provides less information at first sight and requires interaction. In small application, like ParkingLot, UML is better because it provides all the information immediately. In bigger applications, like CoCoME, AIVA is better because high number of connection lines in UML diagrams makes it harder to 1) read types and names of ports and 2) visually trace the relationships.
2. **Which approach provides less complex diagram.** The number of nodes and lines suggests that AIVA is less complex and thus better readable, as it contains a half of graphical elements of UML diagram.
3. **Which approach provides better readable structure/hierarchy.** The spatial composition of these nodes is also better in AIVA, where the hierarchy of the application is easily readable, while RSA (UML) offers only vague idea of the structure.

9.2.3 Generalization

This case study can be generalized for all component models and any UML tool as discussed below. As mentioned in the introduction, any component model can have several features that require extension of UML through profiles to enable modeling of its advanced features. Our OSGi profile is still one of the simplest ones. We used only one type of interface and ports meant for either provided or required packages thus one did not have to pay much attention what type of port or interface it is.

In reality, profiles might be much more complicated according to the complexity of the particular component model. For example an official profile for CORBA Component Model [37] defines seven different types of ports. This suggests that if AIVA managed to get better results against the simple

OSGi profile, it should provide better results for more complex component models as well.

RSA is an advanced commercial tool that contains a lot of rich features to get the information about relationships of elements as soon as possible and supports easy development of both UML profiles and component diagrams. Given that AIVA provides better results against this state of the art UML tool we conjecture that it provides better results against any other tool.

9.3 User Study on User Performance

The main goal of this study is to evaluate performance of users during architecture analysis in two different approaches – UML and AIVA. AIVA is based on the idea that interactivity is beneficial. This idea can be formulated into the following hypothesis in the context of this study: "It is easier and faster for engineers to study structure of component-based applications interactively rather than using static diagrams."

The results of this user study will therefore either confirm or disconfirm this general hypothesis and can help in finding out to what degree is interactivity useful. These questions are important because the level of interactivity used in AIVA is high and could negatively affect the user performance while he collects some more detailed information, namely because a lot of these information is hidden and revealing them requires some activity of the user.

The set of tasks used in the study simulates the activities performed during one step of architecture analysis. These tasks are focused on collecting the knowledge about one component – its features, dependencies and overall context consisting of related components. When analyzing the whole architecture, one needs to repeat this step for most of its components. The concrete set of tasks will be discussed thoroughly further in following subsections.

9.3.1 Design of the study

Profile of Participants

The structure of component-based applications is studied by software engineers who work on these applications. They have a deep knowledge of components and UML to be able to understand to the diagram presented. Such people are hard to get to participate in a user study that takes at least one hour, thus we decided to ask our colleagues to participate. Use of academics and Ph.D. candidates was encouraged by Sensalire et. al. in [50], based on their lessons learned.

All participants were young software engineers with good knowledge of UML and confident in most UML diagrams. Their knowledge of UML component diagram was tested especially before participation. Most of the participants use components on daily basis and the rest were briefly trained before the study. All of the participants were confident in the required basics of component-based development before participation.

All participants were also trained in both tools that were used to test the two approaches. First the tool was presented to them. We shared our working experience on how to get various types of information effectively. Then every participant had an unlimited time to test all types of tasks that he could meet. Participants were handled individually and guidance was given when asked. The training ended when the participant felt confident and able to perform all types of tasks used in this user study.

How the Study Was Performed

The process repeated with every participant was as follows:

1. Verification of knowledge
2. Training in Tool 1
3. Performing all tasks in Tool 1
4. Training in Tool 2
5. Performing all tasks in Tool 2

Verification of UML and component knowledge took about 20 minutes to ensure the participants expertise. Training in both tools took about 40 minutes, until the participant felt confident. All tasks were performed in under 10 minutes, because the tasks were quite short.

All participants were observed for the whole time of the study and there were no interruptions nor any advices while they searched for the answers on given task. The time was measured from the moment the question was read and the understanding was confirmed by the participant, to the point when a correct and full answer was given. Moreover we required users to visually verify the information. Therefore all participants were required to pinpoint the found information in the diagram, a simple answer was not enough.

AIVA was tested first in half cases and UML in the other half. The set of tasks was identical for both approaches however most of these tasks have complex answers – a list of elements or components. Tested approaches provide so different diagrams that the participants could not gain advantage of performing the same tasks again in the other tool. Above that the training in the second tool was performed in the mean time to distract participants and to ensure that he would not gain any advantage. All types of tasks were tested by the participants on different components in a completely different application in the training phase to ensure that they could not gain any knowledge related to the tasks.

9.3.2 Technical details of the study

Tasks

The tasks described below were tested on the CoCoME application. Because of this they are formulated directly for its components, however they can be easily generalized. The tasks are basic and contribute to answer one complex question – how is a particular component (*cocome-osgiDS-store.impl*) integrated in the CoCoME application. One has to find out what this component offers and requires and uncover the ties to other components, simulating the activities performed during one step of architecture analysis. The most complex component of the application was chosen for these tasks.

The tasks were identified according to our experience with the structure of component-based applications and on hints obtained when we were conducting an interview with several software engineers from local software companies. The exact wording was then designed to cover all aspects of one concrete component.

- Q1. Which packages are imported by component *cocome-osgiDS-store.impl*?
- Q2. Which elements of component *cocome-osgiDS-store.impl* are unused (have no relationship)?
- Q3. Which components use the service *StoreIf* provided by *cocome-osgiDS-store.impl*?
- Q4. Which components depend on *cocome-osgiDS-store.impl*?
- Q5. Which components are required by *cocome-osgiDS-store.impl*?
- Q6. Which elements does *cocome-osgiDS-store.impl* need from *cocome-osgiDS-data*?

Hardware

Computer hardware did not influence the results of the study since the bottleneck for performance was user's ability to interact and read the information from diagram. However, to provide complete technical background here is the specification of the testing computer: Intel Core i5 3Ghz CPU, 4GB DDR3 1066Mhz RAM, 7200RPM HDD and most importantly 24" LCD with 1920x1080 resolution. This computer proved to be fast enough to ensure comfortable working experience and the screen resolution was sufficient for visualization purposes.

9.3.3 Results

This section provides detailed results of this study, how the results were measured separately and comparison of these results. For each approach, detailed data as well as statistics are supplied. As the reader may note, the results differ greatly based on a participant. This was caused by individual perception, orientation abilities and how quickly they were able to click with mouse. (A lot of care was paid to prepare all of them enough for upcoming tasks, see Section 9.3.1.)

Twelve users participated in the study, identified as A-L in the tables below. The last two participants (K and L) are the co-authors of this paper and mentored the rest of the participants. Our performance is listed in the results to show the peak performance of the tasks as we exactly knew what we are looking for and how to retrieve these information. We followed the same rules as any other participants and accepted the answer only after visual confirmation. Our results are not used in later statistics.

Performance in AIVA

Results of all participants are presented in Table 9.1 while statistical values are in Table 9.2. The "Total" column in Table 9.2 shows the sum of the measured values in that table, which corresponds to the total experiment performance of a fictive "average," "slowest" (for the "Max" measure) etc. participant; similarly for Table 9.4 further below.

The biggest strength of AIVA was the search for unused elements (Q2) as it provides the answer immediately and most participants were able to read it right away; however, a few of them did not at first understand this information. The biggest weakness was finding the depending components (Q4) because most of the participants forgot to read the type of the arrow indicating the type of the connection and ended a little confused – the time needed for a correct answer was longer.

ID	Q1	Q2	Q3	Q4	Q5	Q6	SUM
A	0:46	0:11	0:20	0:32	0:33	0:28	2:50
B	0:16	0:09	0:42	0:25	0:35	0:25	2:32
C	0:22	0:08	0:18	1:02	0:23	0:27	2:40
D	0:51	0:33	0:20	0:41	0:44	0:50	3:59
E	0:12	0:10	0:11	1:20	0:22	0:10	2:25
F	0:25	0:09	0:23	0:27	0:31	0:38	2:33
G	0:23	0:22	0:19	0:40	0:23	0:29	2:36
H	0:29	0:06	0:16	0:37	0:29	0:16	2:13
I	0:07	0:04	0:08	0:24	0:16	0:24	1:23
J	0:15	0:24	0:17	0:31	0:25	0:17	2:09
K	0:08	0:03	0:08	0:13	0:19	0:10	1:01
L	0:12	0:04	0:08	0:15	0:17	0:11	1:07

Table 9.1: Results of users in AIVA

Measure	Q1	Q2	Q3	Q4	Q5	Q6	Total
Avg	0:24	0:13	0:19	0:39	0:28	0:26	2:32
Med	0:22	0:09	0:18	0:34	0:27	0:26	2:18
Min	0:07	0:04	0:08	0:24	0:16	0:10	1:09
Max	0:51	0:33	0:42	1:20	0:44	0:50	5:00
StdDev	0:13	0:08	0:08	0:17	0:07	0:10	N/A

Table 9.2: Statistics of users in AIVA

Performance in RSA

Results of all participants are presented in Table 9.3 while statistical values are in Table 9.4. The biggest strength of RSA was looking up service clients (Q3) as it provided the answer almost immediately, while the biggest weakness was finding the depending components (Q4) due to worse RSA support in connecting components through ports. Participants gave stable performance as they are familiar with UML notation. The graphical user interface of RSA was more user friendly which also helped users in orientation. Participants were most of the time delayed by accidental clicking on the connection line – RSA had centered screen on it and they lost the context of studied component.

Comparing the Results

A useful information for this study is the performance ratio of AIVA to UML. Comparing this ratio for every participant can bring more insight

ID	Q1	Q2	Q3	Q4	Q5	Q6	SUM
A	1:04	2:40	0:12	2:56	2:43	2:10	11:45
B	1:22	2:06	0:11	1:40	2:39	2:08	10:06
C	1:13	2:30	0:25	1:58	2:49	2:14	11:09
D	1:19	1:30	0:27	1:23	2:25	2:10	9:14
E	0:36	0:59	0:17	0:43	1:41	1:00	5:16
F	1:24	1:05	0:21	1:01	1:40	2:49	6:12
G	0:43	0:30	0:07	0:39	1:46	1:20	5:05
H	0:46	1:14	0:09	0:54	2:17	0:52	6:12
I	0:52	0:34	0:08	0:28	1:00	0:36	3:38
J	0:59	1:06	0:21	0:40	1:48	1:28	6:22
K	0:32	0:42	0:10	0:34	1:07	0:46	3:51
L	0:39	0:52	0:11	0:25	0:54	0:39	3:40

Table 9.3: Results of users in RSA

Measure	Q1	Q2	Q3	Q4	Q5	Q6	Total
Avg	1:01	1:25	0:15	1:14	2:04	1:40	7:42
Med	1:01	1:10	0:14	0:57	2:02	1:48	7:14
Min	0:36	0:30	0:07	0:28	1:00	0:36	3:17
Max	1:24	2:40	0:27	2:56	2:49	2:49	13:05
StdDev	0:16	0:43	0:06	0:43	0:33	0:41	N/A

Table 9.4: Statistics of users in RSA

than comparing the global numbers. The highest ratio had participant A, who was 4,15 times faster in AIVA than in UML. The lowest ratio had participant G, who was only 1,96 times faster in AIVA than in UML. The rest of participants were within these extremes, however they were on average 3 times faster in AIVA – the average test time was 462 seconds, compared to 152 seconds in AIVA.

The average results are compared with standard deviation in Figure 9.7. Normal distribution says that 70% of users would fall within these limits. This figure clearly states that AIVA was faster in tasks Q1, Q2, Q4, Q5 and Q6, that is in 83% of cases, while it was slower in task Q3, which was the strongest task in RSA.

Figure 9.8 comprehensibly presents minimum, maximum and median values in a comparable way, so that these values can be conveniently studied in one place.

Lastly, Figure 9.9 compares the longest times measured in AIVA with shortest times measured in RSA. This figure presents a different look on these ex-

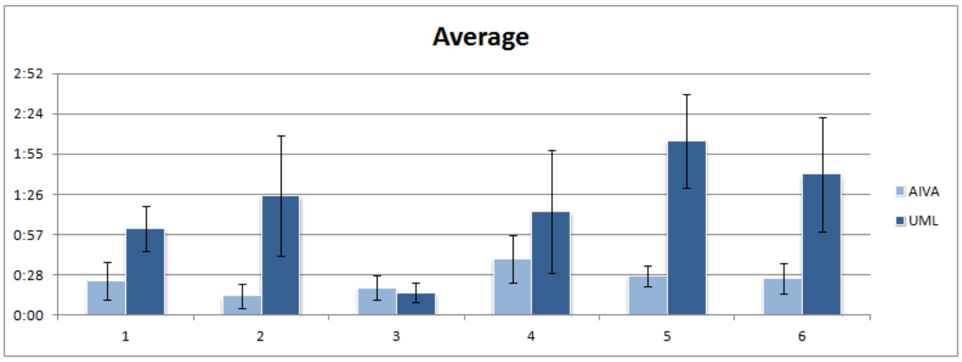


Figure 9.7: Comparison of average results

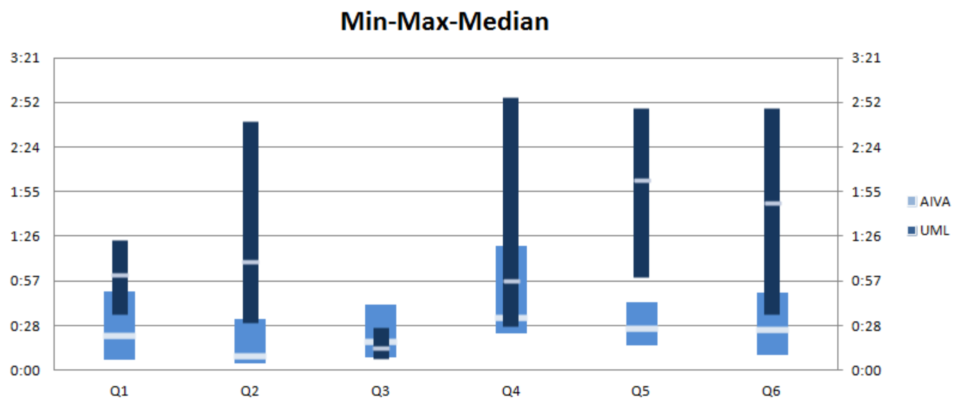


Figure 9.8: Minimum and maximum extreme values with median marked

treme values, testing how a poor use of AIVA compares with best-performing UML users. The numbers show that even in this case, AIVA is comparable in two thirds of tasks; RSA has its best results significantly faster in tasks related to service dependencies.

Two scenarios were tested to compare results in more depth. First scenario tested if users who perform best in UML are also very fast in AIVA. All participants were ordered from fastest to slowest in both AIVA and UML and their order was compared. Four participants from UML top 5 were also in AIVA top 5. The notes on remaining participant who did worse in AIVA showed that he was overconfident because of his expertise in UML. He started the test in AIVA and had to think longer how to finish given tasks. As a result of this scenario, it is possible to conclude that good analysts will benefit from using AIVA.

Second scenario tested where in the distribution are users who were really slow in UML. All participants were ordered from fastest to slowest in UML and also, their performance ratio between AIVA and UML was ordered from

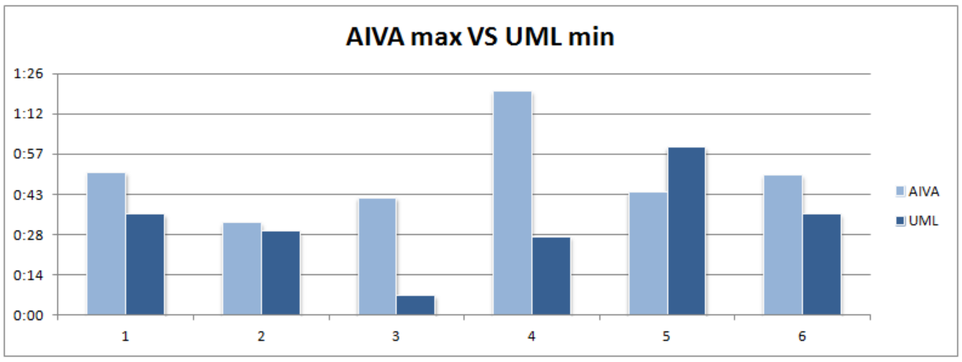


Figure 9.9: Comparison of maximal AIVA results with minimal RSA results

highest to lowest. Four participants from the bottom 5 in UML were in the performance ratio top 5 users. (The one who was not in top 5 had also significantly worst results in AIVA – he also felt confident in AIVA although it turned out he should have kept training for some more time.) These five slowest participants in UML were on average 3,5 times faster in AIVA, while top 5 UML participants were on average only 2,5 times faster. It is possible to conclude that casual users of UML will benefit the most from using AIVA.

9.3.4 Discussion

The previous section provided results of this user study and compared them. By studying these results it is possible to conclude several things on which this section comments.

Measures of AIVA Performance

First of all, the time required to finish a task in AIVA is more consistent – the standard deviation in RSA (48 seconds) is almost four times higher that that of AIVA (14 seconds). The reason is that UML itself and thus also RSA has different levels of recognition and therefore handling for different elements – working is really fast with some (tasks that depend mostly on interfaces) but slow with other ones (tasks that depend mostly on ports). On the other hand, AIVA provides the same level of support for all types of elements on both visual and interaction level.

The previous conclusion leads to a more important one – the choice of tasks is not so important for AIVA as it is for UML. In other words, AIVA should be able to provide stable user performance for any task set, in any component model. In contrast, user's performance in UML depends on selected tasks, selected UML tool and component model.

Task Q4 (searching for clients of the studied component) was the slowest one in both approaches. The reason is that the component was widely used by many other components in the CoCoME application. Therefore it took time to find them all.

One should also look at the fastest task for UML – Q3, which worked with dependencies of interfaces in a tool which is able to list all these dependencies at once. This can be recognized as a best case UML scenario with the fastest time of 7 seconds while AIVA required 8 seconds. Median values are 14 seconds for UML and 18 seconds for AIVA – that is, AIVA is 28% slower. The worst case scenario for UML would be Q5 then, which worked a lot with dependencies of packages (ports). Fastest user finished this task in 16 seconds in AIVA but it took 60 seconds in UML. Median values are 27 seconds in AIVA and 122 seconds in UML – UML is 450% slower. These numbers again indicate that AIVA would be faster in any mixed task set.

From the results provided it is also possible to conclude, that the level of interactivity used in AIVA is useful. This interactivity helped AIVA to provide simpler diagram so users could orientate themselves easier. The overall user performance was better even when the interaction was required to gather the necessary information.

Participant Opinions

This user study confirmed that AIVA is faster than RSA, but we also asked few subjective questions to the participants after they finished:

1. Do you consider AIVA or UML diagram clearer? Why?
2. Was it more comfortable to work in AIVA or RSA? Why?
3. Do you have other suggestions?

All participants answered that AIVA provides clearer diagram that is better readable and understandable. They mentioned these reasons: lesser number of lines, hidden details on zoom, all information in one place and well readable structure of elements.

One participant felt more comfortable in RSA because labels were always visible and click on lines centered the screen. The rest of participants felt more comfortable in AIVA, giving these reasons: clearer GUI, packages shown inside components, much faster operation, information easier to reach, better interactive overview. These participants also did not like the feature that centered the screen after they clicked on the line because it happened often by accident and they lost context.

9.4 Conclusion of Evaluation

This chapter described two studies that should evaluate our novel visualization approach called AIVA.

The research questions asked in the case study were chosen to find out which one of these approaches generates better diagrams in terms of understanding, better structure and lower complexity. As a case examples that presented features of these approaches on differently complex applications were used a simple application with 6 components and CoCoME for complex application with 37 components. The results suggest that AIVA is better almost in all relevant criteria because it generates better understandable, more clearly structured and less complex diagrams than UML does.

The data obtained in the user study show that users working interactively (i.e. in AIVA) are approximately three times faster than those using UML. In only one of six tasks was UML faster, while AIVA performed better in the remaining 5/6 of tasks. The discussion section above provides insight into the reasons and on how different tasks could affect the overall performance. Results of this user study therefore confirm that advanced visualization of component-based application architecture using a high level of interactivity is beneficial for users. Even the increased interaction required to uncover hidden information does not caused significant problems to the users.

Based on these results we want to conclude – AIVA is an approach that is able to provide better results than classic UML diagram can in the field of component-based visualization.

Chapter 10

Conclusion

This chapter summarizes the results of this dissertation thesis. As the key contribution, a new visualization approach called AIVA was created. It uses the ENT meta-model to describe any component application with all important details and visualize them in a graph diagram. It uses a lot of interaction techniques to provide only the information the user wants faster than other approaches. All thesis goals were achieved by developing AIVA, an implementation of proposed solution was completed and evaluated to prove that this approach works. The following subsections will conclude this thesis from several different points of view.

10.1 Evaluation of Thesis Goals

The goals of this thesis were defined in the Section 1.2. These goals are:

1. Visualize the structure of any component-based application as a graph diagram.
2. Visualize a sufficient amount of detail.
3. Provide ways to filter these details and work on different levels of detail interactively.
4. Maximize the advantages of interaction to boost the learning process.

To provide an evaluation of these goals, we would like to discuss how we addressed and fulfilled them. Such discussion is provided below, keeping the order of the original goals.

1. AIVA uses a graph diagram to represent the structure of component-based applications. AIVA uses the ENT meta-model to describe such

structure and uses this description independently on the component model. The ENT meta-model can describe any component-based application – therefore AIVA can visualize the structure of any component-based application.

2. The ENT meta-model can be used to describe which information are important in the context of one component model. The ComAV can be extended by loader plug-in that creates such detailed description. Therefore even if one does not agree that implemented loaders provide sufficient amount of detail, it is possible to create new loaders. Such new loaders would undeniably provide a sufficient amount of detail – based on anyone's definition of word "sufficient".

3. AIVA uses a very strong feature called Category Set that helps it to group elements based on their characteristics and to filter unwanted details away. Category Sets can be changed interactively as AIVA use the whole application model and not only "what is visualized" model. By changing Category Sets one can change the diagram level of details. Moreover, AIVA introduced so called Simple Structure View that hides all the details away and show only the structure.

4. Interaction techniques are used to lower the overall complexity of the diagram – all unnecessary connection lines, connection line labels, interface nodes, element and component related details are hidden and accessed through interaction. AIVA provides any information on one click (or by simply hovering with mouse pointer over an element). The case-study and user-study proved that using AIVA is faster than using current state of the art – UML component diagram. Therefore when AIVA is faster, the learning process should be also faster.

10.2 Current State of Work

Currently the AIVA and ComAV platform are implemented and published under GNU-GPL license. Both these tools are part of the output of this thesis and are functional and ready for use. To conclude the current state of work we provide a chronologically ordered list of what was achieved:

- The ENT meta-model was extended to support more than detailed description of one component [57]. It is now able to describe the structure of the whole component-based application. Moreover it supports hierarchical component applications.
- The ENT meta-model was implemented using model-driven development [56]. A MOF model was created from it's mathematical description, which was used to automatically generate the ENTMM plug-in.

- The idea of rich interactive visualization was introduced [58] – forming into the AIVA in the process of development.
- The ComAV platform was designed and implemented [59]. It offers more than back-end for the AIVA as it can be easily extendable and support various other visualization styles. Reverse-engineering plug-ins were implemented to get the structure of OSGi, EJB and SOFA 2 applications.
- The AIVA itself was evaluated using case-study [60] and user-study. Such evaluation should provide a sufficient argument that AIVA is a working example of how could be the structure of component-based applications visualized better.

10.3 Future Work

The visualization approach itself is finished as it was intended. Future work should address the usability issues, that could help AIVA to be used outside of the research environment. It is important to note, that the current implementation is merely a proof of concept that AIVA could provide a better user experience than UML can.

The work that should be done is mostly in better implementation – not using third party libraries (like JGraphX), reworking the GUI to be more intuitive, not using cross-platform solutions (SWT-AWT), polishing node and line placement, etc. The current implementation is the best that could be achieved given the resources we had.

However, future work should mainly aim on improvements based on the feedback of professional users – as it would improve this approach greatly. Such analysis and deployment in the industry was out of scope of this thesis as it takes years, volunteers and a lot of money to perform it.

10.3.1 Future Research Challenge

We compared UML with AIVA throughout the whole thesis to show that it would require a different approach to solve research challenges we were focused. We succeeded to show that AIVA can provide better results by using interaction techniques and that the ENT can provide more information and use semantics. Another research challenge would be how to use this success and improve UML. UML meta-model could be changed to offer more like the ENT meta-model – the challenge is how much more it could offer if it needs to remain general. UML diagrams could keep their static look only for the static use like image or printer and start to use interactivity in its core specification. If the interactivity features will be clearly defined in its specification all UML tools will have to conform to them – the challenge would be how much to define to ensure some level of user experience but still keep some freedom to UML tool vendors.

Another big research challenge lies in visualization of extra-functional properties. We did a brief study of usage of EFP in AIVA in [55]. The result was that AIVA is able to visualize EFPs but it would require further research in order to offer better usability. It would also require to solve how to use some existing EFP solutions that would be smarter than one time reverse-engineering. Both these problems would bring a new research challenge.

We also tried to bring a touch interaction to AIVA as it would open new ways to visualization of software structure. Touch is another sense that could bring in new possibilities. However, touch interaction in such domain was a big challenge not directly in the scope of our work. We therefore rather focused on our main challenges that needed to be solved.

Bibliography

[1] Felix Bachmann et al. Software architecture documentation in practice: Documenting architectural layers. Special Report CMU/SEI-2000-SR-004, SEI CMU, 2000.

[2] Steffen Becker, Heiko Koziolek, and Ralf Reussner. The palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3 – 22, 2009. Special Issue: Software Performance - Modeling and Analysis.

[3] Premek Brada. The ENT Meta-Model of Component Interface, version 2. Technical report DCSE/TR-2004-14, Department of Computer Science and Engineering, University of West Bohemia, 2004.

[4] Premek Brada. The CoSi Component Model: Reviving the Black-box Nature of Components. In *Proceedings of the 11th International Symposium on Component-Based Software Engineering*, CBSE '08, pages 318–333, Berlin, Heidelberg, 2008. Springer-Verlag.

[5] Premek Brada. A look at current component models from the black-box perspective. *Software Engineering and Advanced Applications, Euromicro Conference*, 0:388–395, 2009.

[6] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quema, and Jean-Bernard Stefani. An Open Component Model and Its Support in Java. In Ivica Crnkovic, Judith A. Stafford, Heinz W. Schmidt, and Kurt Wallnau, editors, *Component-Based Software Engineering*, volume 3054 of *Lecture Notes in Computer Science*, chapter 3, pages 7–22. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2004.

[7] Martin Buchi and Wolfgang Weck. A plea for grey-box components. Technical report, Turku Center For Computer, 1997.

[8] Tomas Bures, Petr Hnetynka, and Frantisek Plasil. SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. In *SERA*, pages 40–48. IEEE Computer Society, 2006.

[9] Heorhiy Byelas, Egor Bondarev, and Alexandru Telea. Visualization of areas of interest in component-based system architectures. In *Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 160–169, Washington, DC, USA, 2006. IEEE Computer Society.

[10] Heorhiy Byelas and Alexandru Telea. Visualization of Areas of Interest in Software Architecture Diagrams. In *Proceedings of the 2006 ACM symposium on Software visualization*, SoftVis '06, pages 105–114, New York, NY, USA, 2006. ACM.

[11] Ivica Crnkovic. Component-based software engineering - new challenges in software development. In *in Software Development. Software Focus*, pages 127–133. John Wiley & Sons, 2001.

[12] Ivica Crnkovic, Michel Chaudron, and Stig Larsson. Component-based development process and component lifecycle. In *Proceedings of the International Conference on Software Engineering Advances*, pages 44–, Washington, DC, USA, 2006. IEEE Computer Society.

[13] Ivica Crnkovic, Michel Chaudron, Severine Sentilles, and Aneta Vulgarakis. A Classification Framework for Component Models. In *Proceedings of the 7th Conference on Software Engineering and Practice in Sweden.* IEEE Computer Society, October 2007.

[14] Stephan Diehl. *Software Visualization - Visualizing the Structure, Behaviour, and Evolution of Software.* Springer, 2007.

[15] Cedric Dumoulin and Sebastien Gerard. Have Multiple Views with one Single Diagram! A Layer Based Approach of UML Diagrams. Research report INRIA-00527850, Institut National de Recherche en Informatique et en Automatique, Universite des Sciences et Technologies de Lille, October 2010.

[16] James Durham. History-making components, April 2011.

[17] Hans-Erik Eriksson, Magnus Penker, Brian Lyons, and David Fado. *UML 2 Toolkit.* Wiley Publishing, Inc., 2004.

[18] Jean-Marie Favre and Humberto Cervantes. Visualization of component-based software. In *Proceedings. First International Workshop on Visualizing Software for Understanding and Analysis*, pages 51 – 60, 2002.

[19] Camilla Forsell. A guide to scientific evaluation in information visualization. In *Information Visualisation (IV), 2010 14th International Conference*, pages 162 –169, July 2010.

[20] David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, pages 1–40. World Scientific Publishing Co., 1992.

[21] Hans Hansson, Mikael Akerholm, Ivica Crnkovic, and Martin Tarngren. SaveCCM - A Component Model for Safety-Critical Real-Time Systems. In *EUROMICRO*, pages 627–635. IEEE Computer Society, 2004.

[22] George T. Heineman and William T. Councill. *Component-based software engineering: putting the pieces together.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[23] Sebastian Herold, Holger Klus, Yannick Welsch, Constanze Deiters, Andreas Rausch, Ralf Reussner, Klaus Krogmann, Heiko Koziolek, Raffaela Mirandola, Benjamin Hummel, Michael Meisinger, and Christian Pfaller. The common component modeling example. In Andreas Rausch, Ralf Reussner, Raffaela

Mirandola, and František Plášil, editors, *The Common Component Modeling Example*, chapter CoCoME - The Common Component Modeling Example, pages 16–53. Springer-Verlag, Berlin, Heidelberg, 2008.

[24] Ric Holt. Software Architecture as a Shared Mental Model. In *Proceedings of International Workshop on Program Comprehension*, 2002.

[25] Lukas Holy, Jaroslav Snajberk, and Premek Brada. Evaluation Component Architecture Visualization Tools. In *Proceedings of International Conference on Information Visualization Theory and Applications.* SciTePress, 2012.

[26] Philip N Johnson-Laird. *Mental models : towards a cognitive science of language, inference, and consciousness / P.N. Johnson-Laird.* Harvard University Press, Cambridge, Mass. :, 1983.

[27] Claire Knight. System and software visualisation. In S.K. Chang, editor, *Handbook of Software Engineering and Knowledge Engineering*, volume II, 2002.

[28] Robert Kosara, Christopher G. Healey, Victoria Interrante, David H. Laidlaw, and Colin Ware. Thoughts on user studies: Why, how, and when. *IEEE Computer Graphics and Applications*, 23:2003, 2003.

[29] Heiko Koziolek. Performance evaluation of component-based software systems: A survey. *Performance Evaluation*, 67(8):634–658, August 2010.

[30] Kung-Kiu Lau and Zheng Wang. Software component models. *IEEE Trans. Software Eng*, 33(10):709–724, 2007.

[31] Jeff McAffer, Jean-Michel Lemieux, and Chris Aniszczyk. *Eclipse Rich Client Platform : Designing, Coding, and Packaging Java Applications.* Addison-Wesley Professional, second edition, April 2010.

[32] D. Mcilroy. Mass-Produced Software Components. In *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*, pages 88–98, 1968.

[33] Philippe Merle and Jean-Bernard Stefani. A formal specification of the Fractal component model in Alloy. Research Report RR-6721, INRIA, 2008.

[34] Joerg Meyer, Jim Thomas, Stephan Diehl, Brian Fisher, and Daniel A. Keim. From Visualization to Visually Enabled Reasoning. In Hans Hagen, editor, *Scientific Visualization: Advanced Concepts*, volume 1 of *Dagstuhl Follow-Ups*, pages 227–245. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2010.

[35] Object Management Group. CORBA Components. OMG Specification formal/02-12-06, Object management Group, 2006.

[36] Object Management Group. Meta Object Facility (MOF) Core Specification. OMG Specification formal/06-01-01, Object management Group, 2006.

[37] Object Management Group. UML Profile for CORBA and CORBA Components Specification. OMG Specification formal/2008-04-07, Object management Group, 2007.

[38] Object Management Group. UML Infrastructure Specification. OMG Specification formal/2009-02-04, Object Management Group, 2009. Version 2.2.

[39] Object Management Group. UML Superstructure Specification. OMG Specification formal/2009-02-02, Object Management Group, 2009. Version 2.2.

[40] OSGi Alliance. OSGi Servise Platform Core Specification. Osgi specification, OSGi Alliance, 2009.

[41] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison Wesley, 1994.

[42] Allan Paivio. *Mental Representations: A Dual Coding Approach (Oxford Psychology Series).* Oxford University Press, USA, September 1990.

[43] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, December 1972.

[44] Jorge Enrique Perez-Martinez. Heavyweight extensions to the UML metamodel to describe the C3 architectural style. *ACM SIGSOFT Software Engineering Notes*, 28(3):5, 2003.

[45] Ana Petricic, Luka Lednicki, and Ivica Crnkovic. Using UML for Domain-Specific Component Models. In *Proceedings of the 14th International Workshop on Component-Oriented Programming*, June 2009.

[46] Frantisek Plasil and Stanislav Visnovsky. Behavior Protocols for Software Components. *IEEE Trans. Software Eng*, 28(11):1056–1076, 2002.

[47] Ruben Prieto-Diaz and Peter Freeman. Classifying software for reusability. *IEEE Software*, 4(1):6–16, January 1987.

[48] Uwe Rastofer. Modelling With Components - Towards a Unified Component Meta-Model. In *12th ECOOP Workshop on Model-based Software Reuse*, 2002.

[49] Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and Frantisek Plasil. *The Common Component Modeling Example: Comparing Software Component Models.* Springer Publishing Company, Incorporated, 1st edition, 2008.

[50] M. Sensalire, P. Ogao, and A. Telea. Evaluation of software visualization tools: Lessons learned. In *Visualizing Software for Understanding and Analysis, 2009. VISSOFT 2009. 5th IEEE International Workshop on*, pages 19 –26, 2009.

[51] Severine Sentilles, Anders Pettersson, Dag Nystrom, Thomas Nolte, Paul Pettersson, and Ivica Crnkovic. Save-ide - a tool for design, analysis and implementation of component-based embedded systems. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 607–610, Washington, DC, USA, 2009. IEEE Computer Society.

[52] Severine Sentilles, Paul Pettersson, Ivica Crnkovic, and J Hakansson. Save-ide: An integrated development environment for building predictable component-based embedded systems. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 493–494, Washington, DC, USA, 2008. IEEE Computer Society.

[53] Mary Shaw and David Garlan. *Software architecture. Perspectives on an emerging discipline.* Prentice Hall Publishing, 1996.

[54] Matti Sillanpaa and Alexandru Telea. Demonstration of the softvision software visualization framework. In *Proceedings 8th European Conference on Software Maintenance and Reengineering*, pages 88–108. IEEE Computer Society, 2004.

[55] J. Snajberk, K. Jezek, and P. Brada. An advanced interactive visualization approach with extra functional properties. In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC*, pages 267–268. IEEE Computer Society, 2012.

[56] Jaroslav Snajberk and Premek Brada. Implementation of a data layer for the visualization of component-based applications. In Dana Pardubska, editor, *Proceedings of the 10th conference on Theory and Practice of Information Technologies (ITAT)*, pages 55–62. PONT s.r.o., 2010.

[57] Jaroslav Snajberk and Premek Brada. ENT: A Generic Meta-Model for the Description of Component-Based Applications. *Electronic Notes in Theoretical Computer Science*, 279(2):59 – 73, 2011. Proceedings of the 8th International Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA).

[58] Jaroslav Snajberk and Premek Brada. Interactive Component Visualization. In *Proceedings of International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 218–225. SciTePress, 2011.

[59] Jaroslav Snajberk and Premek Brada. COMAV - A Component Application Visualisation Tool. In *Proceedings of International Conference on Information Visualization Theory and Applications*. SciTePress, 2012.

[60] Jaroslav Snajberk, Lukas Holy, and Premek Brada. AIVA vs UML: Comparison of Component Application Visualizations in a Case-Study. In *Proceedings of 16th International Conference on Information Visualization*. IEEE Computer Society, 2012.

[61] Jaroslav Snajberk, Lukas Holy, Kamil Jezek, and Premek Brada. An Advanced Interactive Visualization Approach for Component-Based Software: A User Study. In *Proceedings of 7th International Conference on Software Engineering Advances*. IEEE Computer Society, 2012.

[62] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework, 2nd Edition*. Addison-Wesley Professional, 2009.

[63] Sun Microsystems, Inc. Enterprise JavaBeans(TM) Specification, Version 3.0. Sun specification, Sun Microsystems, Inc., May 2006.

[64] Clemenz Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, 3rd edition, 2002.

[65] Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, January 2009.

[66] Alexandru Telea and Lucian Voinea. A Framework for Interactive Visualization of Component-Based Software. In *Proceedings of the 30th EUROMICRO Conference*, pages 567–574, Washington, DC, USA, 2004. IEEE Computer Society.

[67] Lukas Valenta and Premysl Brada. OSGi Component Substitutability Using Enhanced ENT Metamodel Implementation. Technical Report DCSE/TR-2006-05, Department of Computer Science and Engineering, University of West Bohemia, 2006.

[68] Anneliese von Mayrhauser and A. Marie Vans. Program comprehension during software maintenance and evolution. *Computer*, 28:44–55, August 1995.

[69] Matthew O. Ward, Georges G. Grinstein, and Daniel A. Keim. *Interactive Data Visualization - Foundations, Techniques, and Applications.* A K Peters, 2010.

[70] Richard Wettel and Michele Lanza. Visualizing software systems as cities. In *In Proc. of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 92–99. Society Press, 2007.

[71] Ji Soo Yi, Youn ah Kang, John T. Stasko, and Julie A. Jacko. Toward a deeper understanding of the role of interaction in information visualization. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1224–1231, 2007.

Appendix A

Author's Publications

The following papers are essential for my work:

1. Šnajberk, J., Holy, L., Jezek, K. and Brada P.: **An Advanced Interactive Visualization Approach for Component-Based Software: A User Study**, Proceedings of International Conference on Software Engineering Advances, Lisbon, Portugal, 2012
2. Šnajberk, J., Holy, L. and Brada P.: **AIVA vs UML: Comparison of Component Application Visualizations in a Case-Study**, Proceedings of International Conference on Information Visualization, Montpellier, France, 2012
3. Šnajberk, J. and Brada P.: **Interactive Component Visualization**, Proceedings of International Conference on Evaluation of Novel Approaches to Software Engineering, Beijing, China, 2011
4. Šnajberk, J. and Brada P.: **ENT: A Generic Meta-Model for the Description of Component-Based Applications**, Electronic Notes in Theoretical Computer Science, 279(2) : 59 – 73, 2011

The following papers were published in conference proceedings:

1. Šnajberk, J., Holy, L. and Brada P.: **Visualization of Component-Based Applications Structure using AIVA**, Proceedings of European Conference on Software Maintenance and Reengineering, Genova, Italy, 2013
2. Šnajberk, J., Holy, L., Jezek, K. and Brada P.: **An Advanced Interactive Visualization Approach for Component-Based Software: A User Study**, Proceedings of International Conference on Software Engineering Advances, Lisbon, Portugal, 2012

3. Holy, L., Šnajberk, J. and Brada P.: **Lowering Visual Clutter of Clusters in Component Diagrams**, Proceedings of International Conference on Software Engineering Advances, Lisbon, Portugal, 2012

4. Šnajberk, J., Jezek, K. and Brada P.: **An Advanced Interactive Visualization Approach with Extra Functional Properties**, Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing, Innsbruck, Austria, 2012

5. Holy, L., Šnajberk, J. and Brada P.: **Visual clutter reduction for UML component diagrams: A tool presentation**, Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing, Innsbruck, Austria, 2012

6. Šnajberk, J., Holy, L. and Brada P.: **AIVA vs UML: Comparison of Component Application Visualizations in a Case-Study**, Proceedings of International Conference on Information Visualization, Montpellier, France, 2012

7. Holy, L., Šnajberk, J. and Brada P.: **Lowering Visual Clutter in Large Component Diagrams**, Proceedings of International Conference on Information Visualization, Montpellier, France, 2012

8. Šnajberk, J. and Brada P.: **ComAV – A Component Application Visualization Tool**, Proceedings of International Conference on Information Visualization Theory and Application, Rome, Italy, 2012

9. Holy, L., Šnajberk, J. and Brada P.: **Evaluating Component Architecture Visualization Tools**, Proceedings of International Conference on Information Visualization Theory and Application, Rome, Italy, 2012

10. Potuzak, T., Lipka, R., Šnajberk, J. and Brada P.: **Design of a Component-based Simulation Framework for ComponentTesting using SpringDM**, Proceedings of Eastern European Regional Conference on the Engineering of Computer Based Systems, Bratislava, Slovakia, 2011

11. Šnajberk, J. and Brada P.: **Interactive Component Visualization**, Proceedings of International Conference on Evaluation of Novel Approaches to Software Engineering, Beijing, China, 2011

12. Potuzak, T., Šnajberk, J., Lipka, R. and Brada P.: **Component-based Simulation Framework for Component Testing using SpringDM**, Proceedings of International Symposium of the Danube Adria Association for Automation and Manufactoring, Zadar, Croatia, 2010

13. Šnajberk, J. and Brada P.: **Implementation of a Data Layer for the Visualization of Component-based Applications**, Proceedings of Conference on Theory and Practice of Information Technologies, Smrekovica, Slovakia, 2010

The following papers were published in journals:

1. Šnajberk, J. and Brada P.: **ENT: A Generic Meta-Model for the Description of Component-Based Applications**, Electronic Notes in Theoretical Computer Science, 279(2) : 59 – 73, 2011

All technical reports from the following list are available on-line at `http://www.kiv.zcu.cz/publications/techreports.php`.

1. Ing. Jaroslav Šnajberk: **Interactive Visualization of Component-Based Applications**, Technical Report DCSE/TR-2011-03.

Appendix B

ENT Models of Component Models

The following sections provide the ENT models for all three component models that we worked with: OSGi, EJB3 nad SOFA2.

B.1 The OSGi Component Model

B.1.1 Component Types

1. **Bundle**
   - **tagset**: symbolic_name, version
   - **T**: { export_packages, import_packages, provided_services, required_services, native_code, require_bundles, required_execution_environment, use_packages}

B.1.2 Trait Definitions

1. **export_packages**
   - **metatype**: package
   - **K**: ({syntax}, {operational}, {provided}, {structure}, {type}, {permanent}, {multiple}, Lifecycle)
   - **tagset**: version, parameters
   - **extent**: many
2. **import_packages**

- **metatype**: package
- **K**: ({syntax}, {operational}, {required}, {structure}, {type},{permanent}, {single}, Lifecycle)
- **tagset**: bundle_symbolic_name, bundle_version, kind, version_range
- **extent**: many

3. **provided_services**

   - **metatype**: interface
   - **K**: ({syntax}, {operational}, {provided}, {item}, {instance},{optional}, {single}, Lifecycle)
   - **tagset**: service_filter
   - **extent**: many

4. **required_services**

   - **metatype**: interface
   - **K**: ({syntax}, {operational}, {required}, {item}, {instance},{optional}, {multiple}, Lifecycle)
   - **tagset**: service_filter, service_arity
   - **extent**: many

5. **native_code**

   - **metatype**: string
   - **K**: ({syntax}, {operational}, {required}, {item}, {instance},{permanent}, {single}, {development, assembly, deployment, runtime})
   - **tagset**: language, osname, osversion, processor, selection_filter
   - **extent**: many

6. **require_bundles**

   - **metatype**: string
   - **K**: ({syntax}, {operational}, {required}, {structure}, {instance}, {permanent}, {single}, Lifecycle)
   - **tagset**: resolution, bundle_version
   - **extent**: many

7. **require_execution_environment**

   - **metatype**: set

- **K**: ({syntax}, {operational}, {required}, {item}, {type}, {permanent}, {single}, {development, assembly, deployment, runtime})
- **tagset**: ∅
- **extent**: many

8. **use_packages**
   - **metatype**: map
   - **K**: ({syntax}, {operational}, {provided_and_required}, {item}, {instance}, {permanent}, {single}, Lifecycle)
   - **tagset**: ∅
   - **extent**: many

B.1.3 Tag Definitions

1. **symbolic_name**
   - **metatype**: string
   - **valset**: ∅
   - **d**: ϵ
2. **parameters**
   - **metatype**: map
   - **valset**: ∅
   - **d**: ϵ
3. **version**
   - **metatype**: versionidentifier
   - **valset**: ∅
   - **d**: 0.0.0
4. **bundle_symbolic_name**
   - **metatype**: string
   - **valset**: ∅
   - **d**: ϵ
5. **bundle_version**
   - **metatype**: versioninterval

- **valset**: $\emptyset$
- **d**: ϵ

6. **kind**
   - **metatype**: enumerator
   - **valset**: static, dynamic
   - **d**: ϵ
7. **selection_filter**
   - **metatype**: string
   - **valset**: $\emptyset$
   - **d**: ϵ
8. **language**
   - **metatype**: string
   - **valset**: $\emptyset$
   - **d**: ϵ
9. **processor**
   - **metatype**: string
   - **valset**: $\emptyset$
   - **d**: ϵ
10. **osname**
    - **metatype**: string
    - **valset**: $\emptyset$
    - **d**: ϵ
11. **osversion**
    - **metatype**: string
    - **valset**: $\emptyset$
    - **d**: ϵ
12. **resolution**
    - **metatype**: enumeration
    - **valset**: mandatory, optional
    - **d**: mandatory

13. **version_range**
    - **metatype**: versioninterval
    - **valset**: $\emptyset$
    - **d**: ϵ
14. **service_filter**
    - **metatype**: hashmap
    - **valset**: $\emptyset$
    - **d**: ϵ
15. **service_arity**
    - **metatype**: enumeration
    - **valset**: 0..1, 1, 0..N, 1..N
    - **d**: 0..N

B.2 The SOFA2 Component Model

B.2.1 Component Types

1. **Architecture**
    - **tagset**: frame, implementation, Version
    - **T**: { provided_interface, required_interface, properties, environment_assumptions}

B.2.2 Trait Definitions

1. **provided_interface**
    - **metatype**: interface
    - **K**: ({syntax}, {operational}, {provided}, {structure}, {type}, {permanent}, {single}, Lifecycle)
    - **tagset**: interface_type, communication_style
    - **extent**: many
2. **required_interface**
    - **metatype**: interface
    - **K**: ({syntax}, {operational}, {required}, {structure}, {type}, {permanent}, {single}, Lifecycle)

- **tagset**: interface_type, communication_style
- **extent**: many

3. **properties**

- **metatype**: string
- **K**: ({syntax}, {data}, {provided}, {item}, {instance}, {permanent}, {single}, Lifecycle)
- **tagset**: service_filter
- **extent**: many

4. **environment_assumptions**

- **metatype**: efp
- **K**: ({extra-functional}, {data}, {required}, {item}, {constant}, {optional}, {single}, {deployment, assembly})
- **tagset**: kind
- **extent**: many

B.2.3 Tag Definitions

1. **communication_style**

- **metatype**: string
- **valset**: $\emptyset$
- **d**: ϵ

2. **interface_type**

- **metatype**: string
- **valset**: $\emptyset$
- **d**: ϵ

3. **version**

- **metatype**: versionidentifier
- **valset**: $\emptyset$
- **d**: 0.0.0

4. **frame**

- **metatype**: string
- **valset**: $\emptyset$

- **d**: ϵ

5. **implementation**
   - **metatype**: string
   - **valset**: $\emptyset$
   - **d**: ϵ
6. **kind**
   - **metatype**: enumeration
   - **valset**: attribute, capacity, maximum, minimum
   - **d**: ϵ

B.3 The EJB3 Component Model

B.3.1 Component Types

1. **SessionBean**
   - **tagset**: state, name
   - **T**: { business_interfaces, business_references, environment_entries, event_publishers, resources, security_roles, web_service_references, web_services}
2. **MessageDrivenBean**
   - **tagset**: description, mappedName, messageListener, name
   - **T**: { business_references, event_publishers, event_listeners, resources, environment_entries security_roles, web_service_references }
3. **Entities**
   - **tagset**: name, table
   - **T**: { entity_properties}

B.3.2 Trait Definitions

1. **resources**
   - **metatype**: attribute
   - **K**: ({syntax}, {data}, {required}, {structure}, {instance}, {permanent}, {single}, {development, assembly, setup, runtime})

- **tagset**: name
- **extent**: many

2. **entity_properties**
   - **metatype**: string
   - **K**: ({syntax}, {data}, {provided}, {item}, {instance}, {permanent}, {multiple}, Lifecycle)
   - **tagset**: id, column, table, arity, persistency
   - **extent**: many
3. **event_publishers**
   - **metatype**: event
   - **K**: ({syntax}, {operational}, {provided}, {item}, {instance}, {mandatory}, {single}, {development, assembly, setup, runtime})
   - **tagset**: type
   - **extent**: many
4. **event_listeners**
   - **metatype**: event
   - **K**: ({syntax}, {operational}, {required}, {item}, {instance}, {mandatory}, {single}, {development, assembly, setup, runtime})
   - **tagset**: type
   - **extent**: many
5. **business_interfaces**
   - **metatype**: interface
   - **K**: ({syntax}, {operational}, {provided}, {structure}, {type}, {mandatory}, {multiple}, Lifecycle)
   - **tagset**: locality
   - **extent**: many
6. **business_references**
   - **metatype**: interface
   - **K**: ({syntax}, {operational}, {required}, {structure}, {instance}, {permanent}, {single}, Lifecycle)
   - **tagset**: $\emptyset$
   - **extent**: many

7. **environment_entries**
   - **metatype**: structure
   - **K**: ({syntax}, {data}, {required}, {item}, {instance}, {permanent}, {single}, {development, assembly, deployment, runtime})
   - **tagset**: value
   - **extent**: many
8. **security_roles**
   - **metatype**: string
   - **K**: ({semantics}, {operational}, {provided}, {item}, {instance}, {permanent}, {single}, {deployment, runtime})
   - **tagset**: service_filter, service_arity
   - **extent**: many
9. **web_services**
   - **metatype**: interface
   - **K**: ({syntax}, {operational}, {provided}, {structure}, {type}, {permanent}, {multiple}, {development, assembly, deployment, runtime})
   - **tagset**: wsdlLocation, endpointInterface
   - **extent**: many
10. **web_service_references**
    - **metatype**: interface
    - **K**: ({syntax}, {operational}, {required}, {structure}, {instance}, {permanent}, {multiple}, {development, assembly, deployment, runtime})
    - **tagset**: wsdlLocation
    - **extent**: many

B.3.3 Tag Definitions

1. **state**
   - **metatype**: enumeration
   - **valset**: stateless, stateful
   - **d**: ϵ
2. **name**

- **metatype**: string
- **valset**: $\emptyset$
- **d**: ϵ

3. **description**
   - **metatype**: string
   - **valset**: $\emptyset$
   - **d**: 0.0.0
4. **mappedName**
   - **metatype**: string
   - **valset**: $\emptyset$
   - **d**: ϵ
5. **messageListener**
   - **metatype**: string
   - **valset**: $\emptyset$
   - **d**: ϵ
6. **table**
   - **metatype**: string
   - **valset**: $\emptyset$
   - **d**: ϵ
7. **table**
   - **metatype**: string
   - **valset**: $\emptyset$
   - **d**: ϵ
8. **id**
   - **metatype**: string
   - **valset**: $\emptyset$
   - **d**: ϵ
9. **value**
   - **metatype**: string
   - **valset**: $\emptyset$

- **d**: ϵ

10. **column**
    - **metatype**: string
    - **valset**: $\emptyset$
    - **d**: ϵ
11. **persistency**
    - **metatype**: enumeration
    - **valset**: persistent, transient
    - **d**: persistent
12. **arity**
    - **metatype**: enumeration
    - **valset**: OneToMany, ManyToOne
    - **d**: ϵ
13. **type**
    - **metatype**: enumeration
    - **valset**: topic, queue
    - **d**: ϵ
14. **locality**
    - **metatype**: enumeration
    - **valset**: local, remote
    - **d**: ϵ
15. **wsdlLocation**
    - **metatype**: string
    - **valset**: $\emptyset$
    - **d**: ϵ
16. **endpointInterface**
    - **metatype**: string
    - **valset**: $\emptyset$
    - **d**: ϵ

Appendix C

Full Sized Images from Comparison Case Study

This chapter contains full sized images of the application ParkingLot that were presented in Section 9.2.

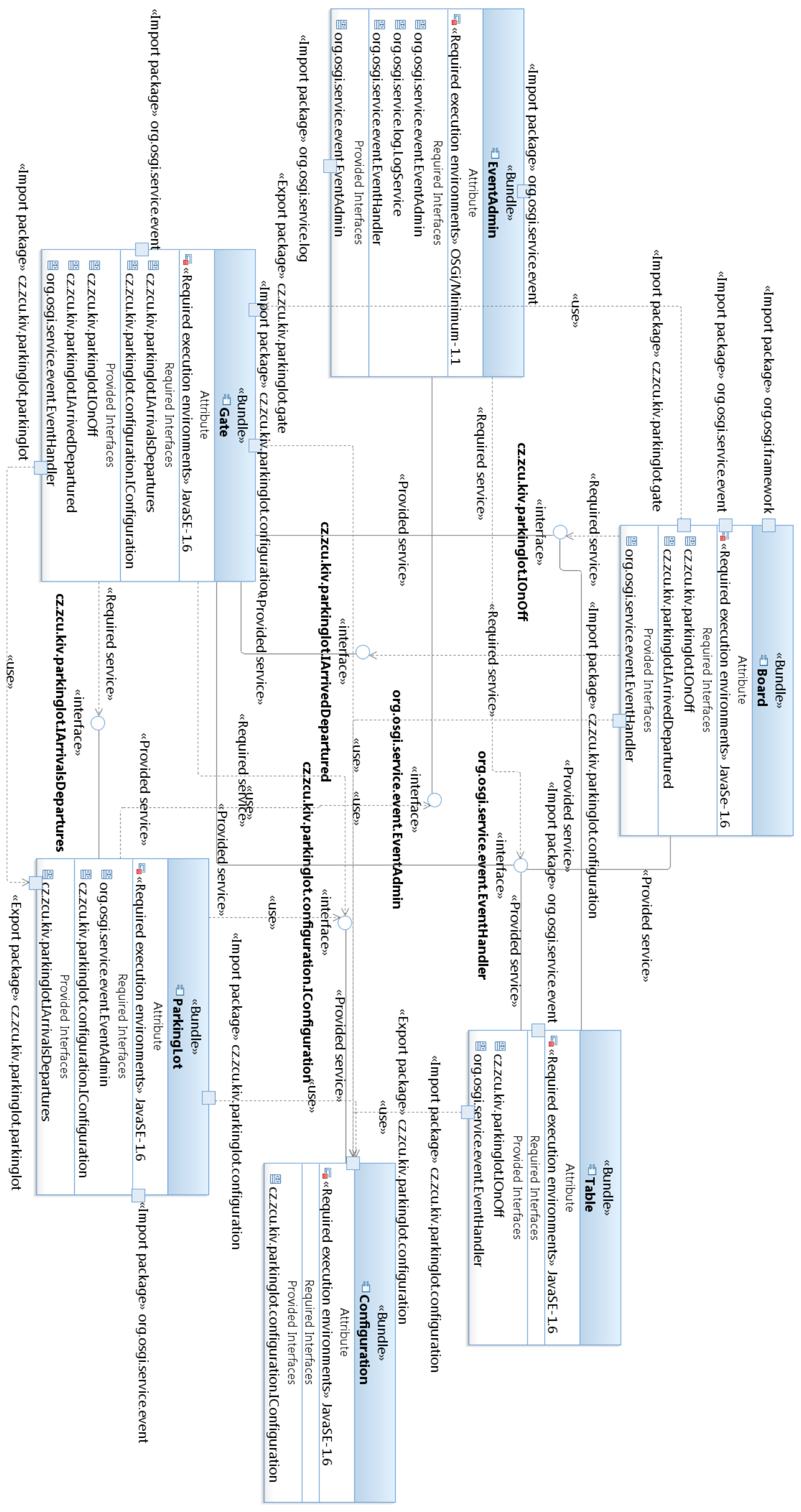

«Bundle» Board
Attribute
«Required execution environments» JavaSE-1.6
Required Interfaces
cz.zcu.kiv.parkinglot.IOnOff
cz.zcu.kiv.parkinglot.IArrivedDepartured
Provided Interfaces
org.osgi.service.event.EventHandler
«Bundle» EventAdmin
Attribute
«Required execution environments» OSGi/Minimum-1.1
Required Interfaces
org.osgi.service.event.EventAdmin
org.osgi.service.log.LogService
org.osgi.service.event.EventHandler
Provided Interfaces
org.osgi.service.event.EventAdmin
«Bundle» Gate
Attribute
«Required execution environments» JavaSE-1.6
Required Interfaces
cz.zcu.kiv.parkinglot.IArrivalsDepartures
cz.zcu.kiv.parkinglot.configuration.IConfiguration
Provided Interfaces
cz.zcu.kiv.parkinglot.IOnOff
cz.zcu.kiv.parkinglot.IArrivedDepartured
org.osgi.service.event.EventHandler
«Bundle» ParkingLot
Attribute
«Required execution environments» JavaSE-1.6
Required Interfaces
org.osgi.service.event.EventAdmin
cz.zcu.kiv.parkinglot.configuration.IConfiguration
Provided Interfaces
cz.zcu.kiv.parkinglot.IArrivalsDepartures
«Bundle» Table
Attribute
«Required execution environments» JavaSE-1.6
Required Interfaces
cz.zcu.kiv.parkinglot.IOnOff
Provided Interfaces
org.osgi.service.event.EventHandler
«Bundle» Configuration
Attribute
«Required execution environments» JavaSE-1.6
Required Interfaces
Provided Interfaces
cz.zcu.kiv.parkinglot.configuration.IConfiguration
«Import package» org.osgi.framework
«Import package» org.osgi.service.event
«Import package» cz.zcu.kiv.parkinglot.gate
«Import package» cz.zcu.kiv.parkinglot.configuration
«Import package» org.osgi.service.log
«Export package» cz.zcu.kiv.parkinglot.gate
«Export package» cz.zcu.kiv.parkinglot.parkinglot
«Import package» cz.zcu.kiv.parkinglot.parkinglot
«Export package» cz.zcu.kiv.parkinglot.configuration
«Export package» org.osgi.service.event
«use»
«Required service»
«Provided service»
«interface»
cz.zcu.kiv.parkinglot.IOnOff
cz.zcu.kiv.parkinglot.IArrivedDepartured
org.osgi.service.event.EventAdmin
org.osgi.service.event.EventHandler
cz.zcu.kiv.parkinglot.configuration.IConfiguration
cz.zcu.kiv.parkinglot.IArrivalsDepartures

Figure C.1: ParkingLot in UML

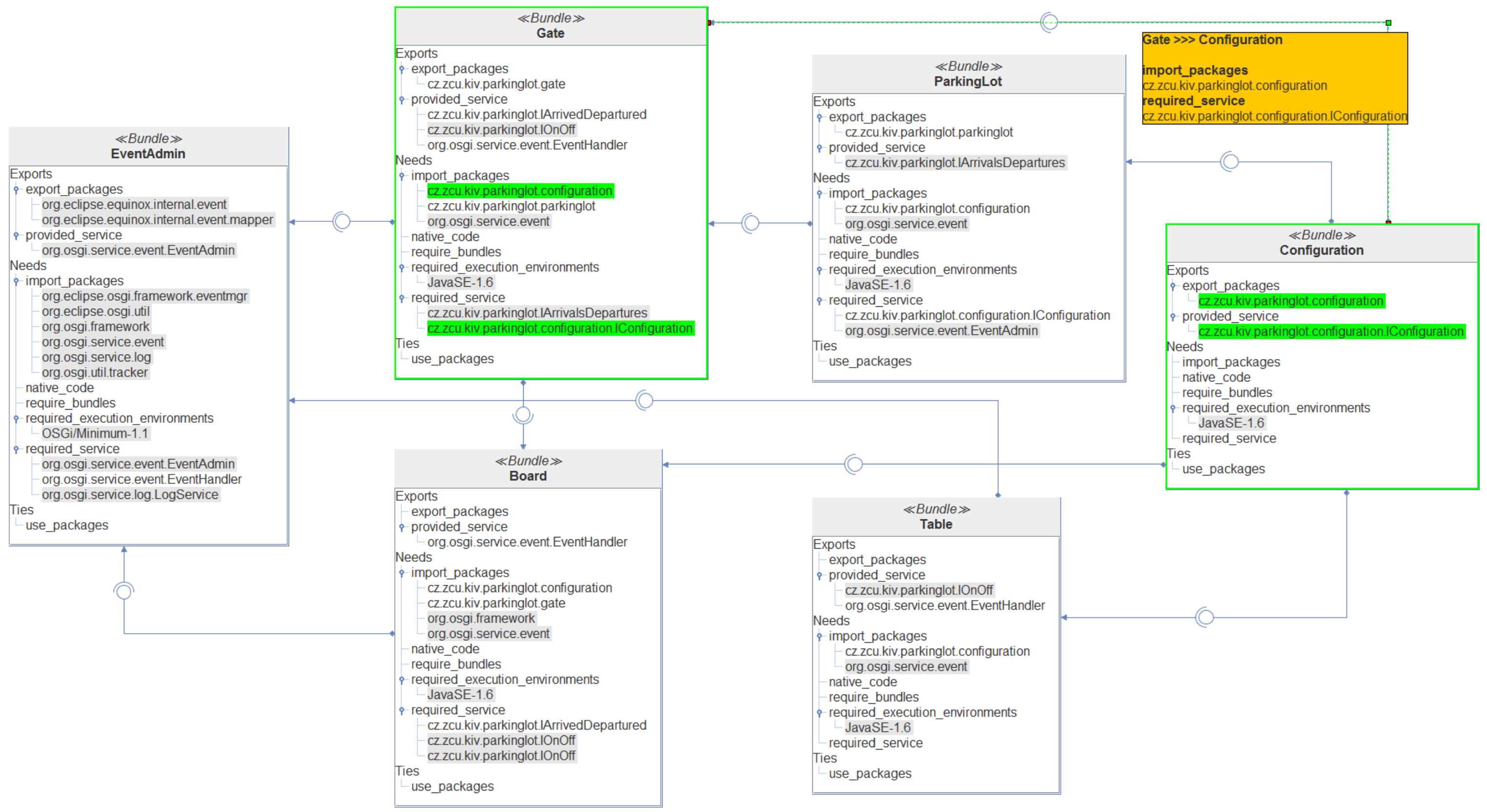

≪Bundle≫
Gate
Exports
export_packages
cz.zcu.kiv.parkinglot.gate
provided_service
cz.zcu.kiv.parkinglot.IArrivedDepartured
cz.zcu.kiv.parkinglot.IOnOff
org.osgi.service.event.EventHandler
Needs
import_packages
cz.zcu.kiv.parkinglot.configuration
cz.zcu.kiv.parkinglot.parkinglot
org.osgi.service.event
native_code
require_bundles
required_execution_environments
JavaSE-1.6
required_service
cz.zcu.kiv.parkinglot.IArrivalsDepartures
cz.zcu.kiv.parkinglot.configuration.IConfiguration
Ties
use_packages
≪Bundle≫
ParkingLot
Exports
export_packages
cz.zcu.kiv.parkinglot.parkinglot
provided_service
cz.zcu.kiv.parkinglot.IArrivalsDepartures
Needs
import_packages
cz.zcu.kiv.parkinglot.configuration
org.osgi.service.event
native_code
require_bundles
required_execution_environments
JavaSE-1.6
required_service
cz.zcu.kiv.parkinglot.configuration.IConfiguration
org.osgi.service.event.EventAdmin
Ties
use_packages
Gate >>> Configuration
import_packages
cz.zcu.kiv.parkinglot.configuration
required_service
cz.zcu.kiv.parkinglot.configuration.IConfiguration
≪Bundle≫
EventAdmin
Exports
export_packages
org.eclipse.equinox.internal.event
org.eclipse.equinox.internal.event.mapper
provided_service
org.osgi.service.event.EventAdmin
Needs
import_packages
org.eclipse.osgi.framework.eventmgr
org.eclipse.osgi.util
org.osgi.framework
org.osgi.service.event
org.osgi.service.log
org.osgi.util.tracker
native_code
require_bundles
required_execution_environments
OSGi/Minimum-1.1
required_service
org.osgi.service.event.EventAdmin
org.osgi.service.event.EventHandler
org.osgi.service.log.LogService
Ties
use_packages
≪Bundle≫
Configuration
Exports
export_packages
cz.zcu.kiv.parkinglot.configuration
provided_service
cz.zcu.kiv.parkinglot.configuration.IConfiguration
Needs
import_packages
native_code
require_bundles
required_execution_environments
JavaSE-1.6
required_service
Ties
use_packages
≪Bundle≫
Board
Exports
export_packages
provided_service
org.osgi.service.event.EventHandler
Needs
import_packages
cz.zcu.kiv.parkinglot.configuration
cz.zcu.kiv.parkinglot.gate
org.osgi.framework
org.osgi.service.event
native_code
require_bundles
required_execution_environments
JavaSE-1.6
required_service
cz.zcu.kiv.parkinglot.IArrivedDepartured
cz.zcu.kiv.parkinglot.IOnOff
cz.zcu.kiv.parkinglot.IOnOff
Ties
use_packages
≪Bundle≫
Table
Exports
export_packages
provided_service
cz.zcu.kiv.parkinglot.IOnOff
org.osgi.service.event.EventHandler
Needs
import_packages
cz.zcu.kiv.parkinglot.configuration
org.osgi.service.event
native_code
require_bundles
required_execution_environments
JavaSE-1.6
required_service
Ties
use_packages

Figure C.2: ParkingLot in AIVA, with one connection (Gate – Configuration) selected