University of West Bohemia in Pilsen
Department of Computer Science and Engineering
Univerzitni 8
30614 Pilsen
Czech Republic

Interactive Visualization of Component-Based Applications

Supervisor: Přemek Brada

Jaroslav Šnajberk

Technical Report No. DCSE/TR-2011-03
May, 2011

Distribution: public

Technical Report No. DCSE/TR-2011-03
May 2011

Interactive Visualization of Component-Based Applications

Jaroslav Šnajberk

Abstract

Component-based software engineering is a mature field of study that enables better reusability of already written code by introducing coarser-grained components that are similar to similar concepts known from other industries. Components enable to create even more complex software systems than before but there is no smart way how to visualize the structure of these systems. It is well known that interactivity is a great help in visualization: it can enhance human performance and speed up the process of understanding the visualized information. However this knowledge lays largely unused by mainstream software engineering research as there is no approach able to visualize structure of these complex software systems, especially one that could offer benefits of interactive visualization. This failure is discussed in this report together with a proposed solution. As a basis of the discussion we provide an overview of basic terms of component-based engineering and introduce details concerned with interactive visualization to emphasize its importance. The lack of sufficient approach is documented by a survey we made and present in this report together with a discussion about state of the art in this area. The proposed solution is a new approach that uses the advantages of interactive techniques for the visualization of component-based applications structure.

The work was supported by the UWB grant SGS-2010-028 Advanced Computer and Information Systems.

Department of Computer Science and Engineering
Univerzitni 8
30614 Pilsen
Czech Republic

Contents

1 Introduction

The overall size and complexity of current software systems is much higher than before. As the performance of computer 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 and thus remain unchanged from 1992, when Garlan [17] 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 [25] firstly mentioned the concept of component. In 1987 [1] 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 be able to introduce 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 [?] in 2004, SOFA [7] in 2006 and CoSi [4] in 2008.

The complexity of software system based on any architecture is increasing in time, so in 1994 the Unified Modeling Language (UML) has taken its initial form to

[1] this paragraph is based on overview from http://www.ibm.com/developerworks/webservices/library/co-tmline/

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

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 case it is a 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 ones 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 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 purposes of 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 these 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 Goal of the Work

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.

In order to achieve the stated goals, it was necessary to extend the ENT meta-model and add features that enable the modeling of relations between components together with hierarchical composition. This extension was the first step that would enable a new visualization approach.

The structure of the report is as follows: Section 2 covers the basic terms of component-based development and discusses the issues of very different understandings of components. Section 3 then provide an introduction to visualization with a focus on the cognitive limitations 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 Section 4, with a deeper description of problems of current approaches. Section 5 provides a thorough description of the ENT meta-model together with its formal specification. Finally, the proposed interactive visualization approach is presented in Section 6, which should outline the scope of a future Ph.D. dissertation.

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 [19] and Crnkovic [10], [11]. 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 implemented 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, however, a very difficult task, because to provide a reusable component, it has to be general, but still it has to avoid overgeneralization 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 [19]:

> 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 needs, which are well-defined and have to be satisfied. One of the consequences is that a component is independent of other components except its child components. When an application should be extended, it doesn't 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 component and what a component is. Before we continue in theory, lets look at a few concrete component descriptions that will help to understand the term.

- **OSGi component** [32] 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 is realized through these services; thus, applications are composed by simply deploying 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 recieve previously registered ones. A service is an instance of a class, that implements interface of the conrete service.
- **SOFA component** [7] 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 lifecycle 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 called a composite component. 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** [2] 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 1) to illustrate the usage of user controls. In the top left corner you can see the toolbox of user controls, 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.

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]:

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

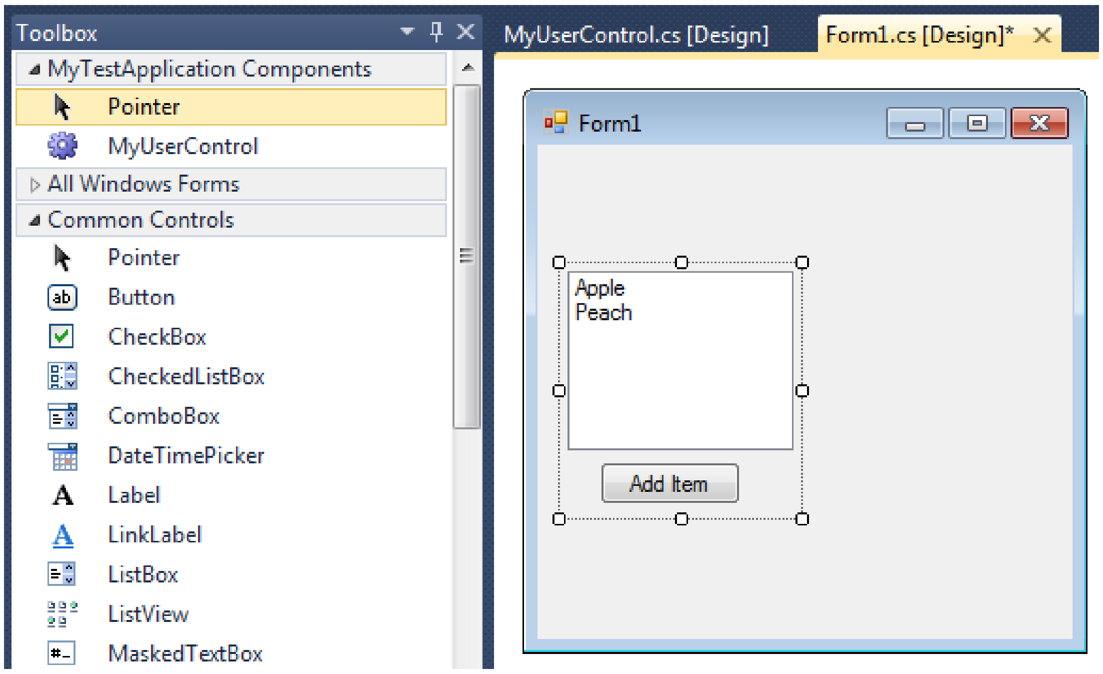


Figure 1: Component development in Visual Studio IDE

A Component is:

- an opaque implementation of functionality
- subject to third-party composition
- conformant with a component model

Szyperski defined a component in [48] 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 [49], 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 with 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 without any danger. 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 doesn't 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 [24].

> 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 [24] and Szyperski [48], 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 [12], 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) [47] 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 type can communicate with what. 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 create 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 real 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 [34], 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 [48].

> 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 [6]. Grayboxes are used, for example, in the SOFA component model [7] in the architecture of frames.

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 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 [52]:

> 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 [13]:

- **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 [3], etc.

- **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.

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

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 [13], Taylor provides a more brief, but also very interesting description of software visualization in [49]. In the scope of this report, 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.

We are not the the only ones who want to address problems of current software visualization approaches. Knight [22] 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.

3.1 Principles for creation of a mental model

Before the analysis of 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 [21].

Ric Holt uses the theory of mental models and apply it to software architectures in [20], 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 [20] identified several laws and principles, from which we will discuss three of the most important laws for 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 us 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 omit 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 aboutthe 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 visual representation should be composed of both these types of information so we can use the maximum capacity of our brains.

3.2 Visually Enabled Reasoning

Meyer et al. define the new science of visually enabled reasoning in [27] 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 3.2.

Visual Analytics itself is a field of information visualization that incorporates human computer interaction (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 user experience and performance it is essential to use the advantages of dynamic interactive principles and adapt them to create a perfect match with

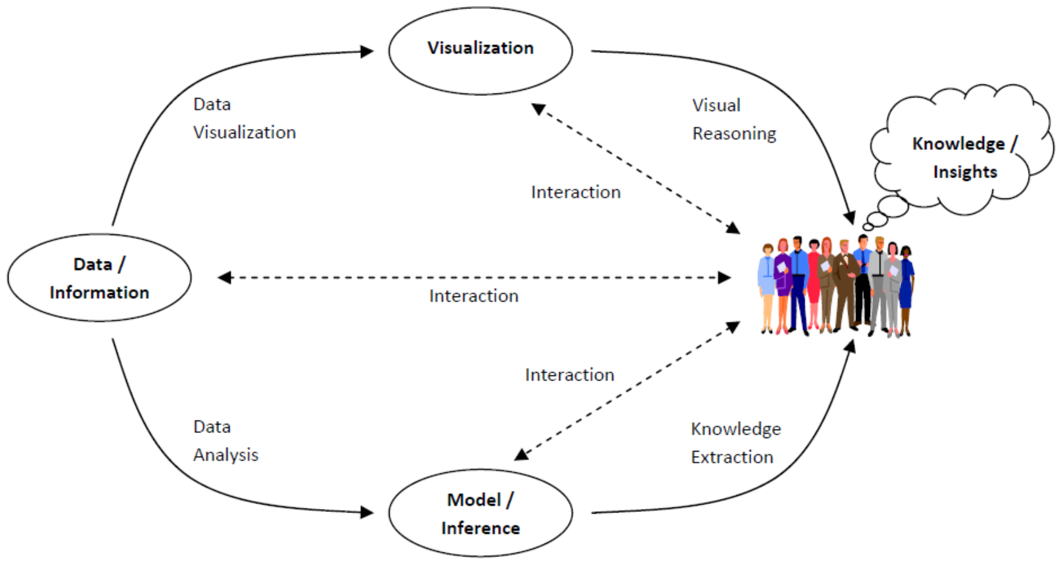


Figure 2: Diagram of visual reasoning [27]

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 [27].

Interactive Reasoning is the process of distinguishing between ideas in order to create new relations and insights based on collected evidence [27]. 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 isn't 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. [27] 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.

3.3 Interactive Visualization

Interaction plays a key role in information visualization and leads to visually enabled reasoning as mentioned in the previous subsection. This subsection provides more understanding about this term and defines the main categories of interactive techniques. These categories are the result of study by Yi et al. presented in [54] and can be considered as key for our future work, because they offer different interaction categories based on user intent.

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.

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 [53]. 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.

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 doesn't 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 don't 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.

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 grepeats.

It is clear that all the interaction techniques can't be combined together, because they could change the same representation of components for different reasons. To enable the maximum potential of interactive component visualization, it is necessary to study different interactive techniques and choose sufficiently different techniques across all categories.

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 [46], CORBA [28] and OSGi [32]. More can be found in commercial applications and even more component models – for example, SOFA [7], Fractal [26] 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, it is still impossible to provide advantages as 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 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[4] is a great example of general "boxes-and-arrows" visualizations. 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 [31] component diagram, which introduces semantics to the visual notation; thus it is not only "boxes-and-arrows".

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

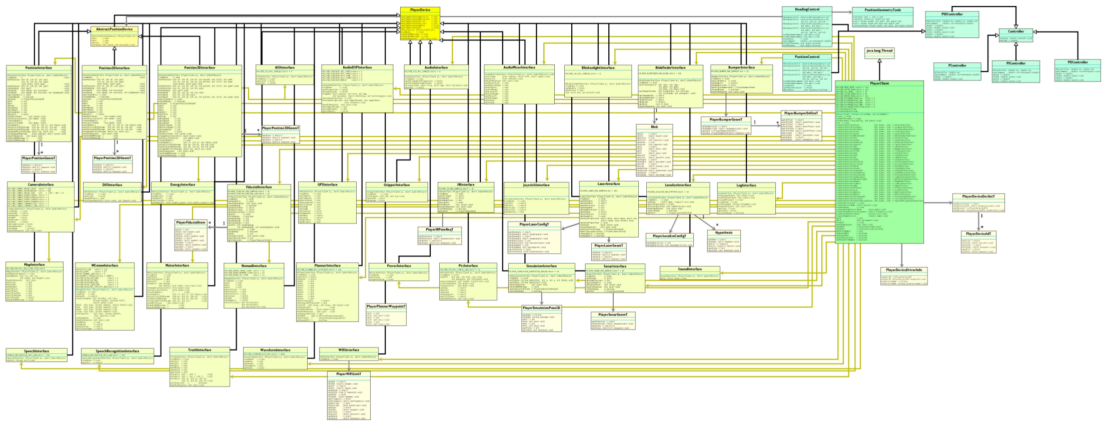


Figure 3: An example UML class diagram of java-player taken from http://java-player.sourceforge.net

The answer for how to visualize a component-based application is the use of some initialization method, which firstly sets the environment for a concrete component model and after that can visualize different applications in a similar way. Favre mentioned the need for such a description of a component model prior to the visualization of a component in [16]. An example of such an initialization method is the use of profiles in UML; more details are in Subsection 4.1.1. In any case, there has to be a way to visualize details bound to a specific component model.

4.1.1 UML 2

As mentioned earlier, UML 2 supports extensions in the form of profiles which can offer a customization of the general "boxes-and-arrows" able to capture enough details about the structure of the application 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 [30] and SaveCCM [36].

The problem is that UML doesn't fulfill some of the needs 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 needs as follows:

- In component-based development, there are roles with very different interests and needs (developer, assembler, etc.). UML uses a diagram for every role in order to provide the exact amount of detail for each of them.

- Stereotypes, which are the power of the UML extension mechanism, behave more like tags – they only say that the attribute or method belongs to some group. But component-based development, because of its diversity, needs a mechanism to model new types of elements apart from attributes and methods. Ideally, the model should provide some meta-information to improve orientation in the elements of the component.

- UML was designed to be static, to show all information at once and provide the same output on both screen and paper. However, when a component assembler works with hundreds of components, he needs to keep orientated in a complex "boxes-and-arrows" diagram, accessing levels of detail on demand interactively.

- Similar to the previous point, but closer to implementation, when acomponent architect looks at the components, he may be interested in the existence of all the elements, but he doesn't want to be bothered with the details about these elements.

- 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 4.1.1. A component diagram is usually as 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 rather 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.

4.1.2 Layered UML

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 slight 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 provide a way that is able to accommodate multiple views in one diagram, so the user can easily add (or hide) details or items or change layouts.

In [14], Dimoulin describes such a feature that can extend UML. Dumoulin decided to choose a layered approach: a final diagram is completed by composing all visible layers together. These layers are rather a change-sets, which says what should be changed on the main layer. For example: Layer one can create all the items without any details, layer two can add details to these items, layer three can color some items to emphasize them, layer four can add comments, etc.

This feature is developed as a part of an open source UML tool named Eclipse Papyrus[5]. An example of how these multiple views look is in Figure 4.

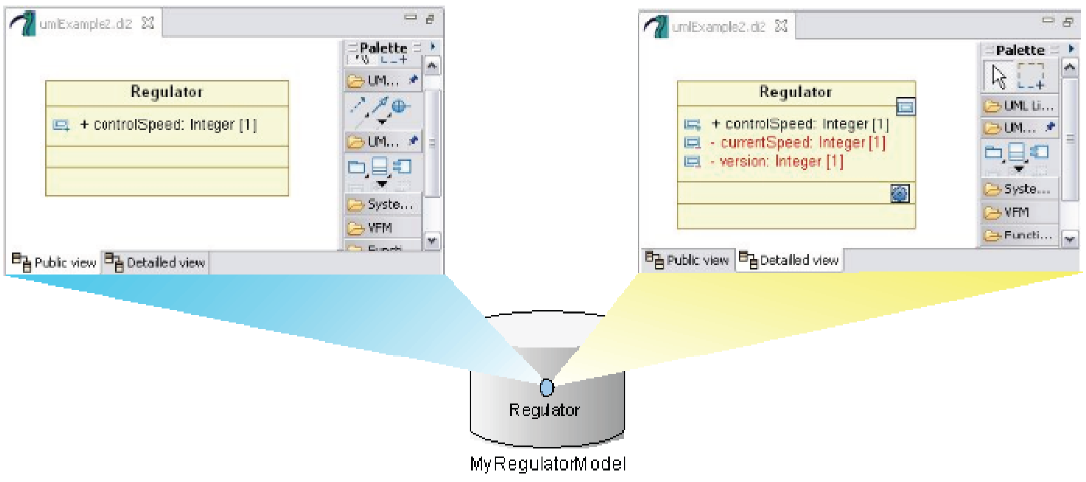


Figure 4: Multiple views in one UML diagram [14]

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:

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

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

4.1.3 Area of Interest in UML

Area of interest is used to highlight a somehow interesting part of an application. Beylas describes in [8] and [9] 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 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.

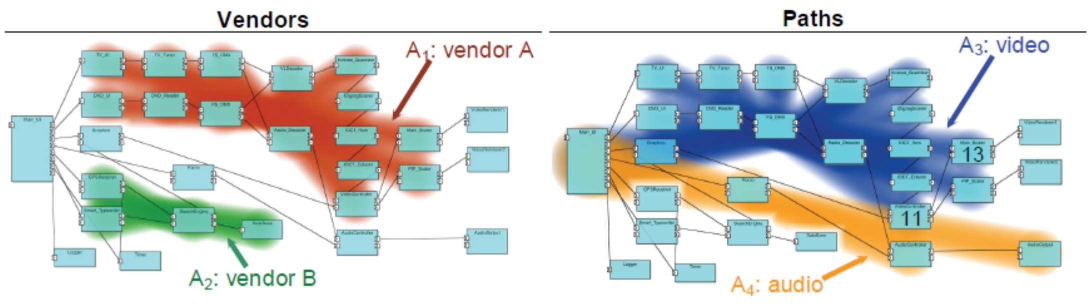


Figure 5: Several Areas of Interest in a component diagram [8]

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.

4.1.4 SoftVision

Telea et al. describe the principles of an interactive visualization framework able to visualize any component-based application in [50]. Later Sillanpaa demonstrate possibilities of this framework named SoftVision in [43]. SoftVision provides the functionality needed to create one's own visualization tool, namely it allows a user to: define a new representation of an item (how items are drawn), define new layouts (where items are drawn), pick a callback 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; this is apparent from its description in [43]:

> 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 [33] 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 6.

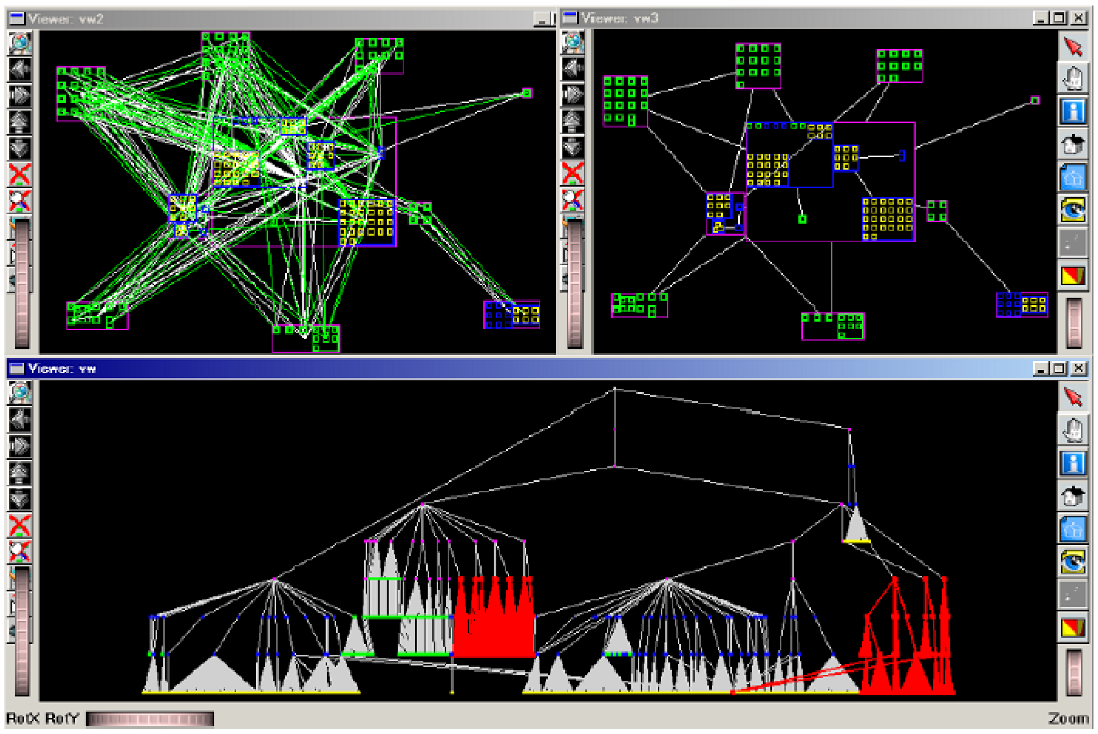


Figure 6: SoftVision visualizing architecture on different layouts [43]

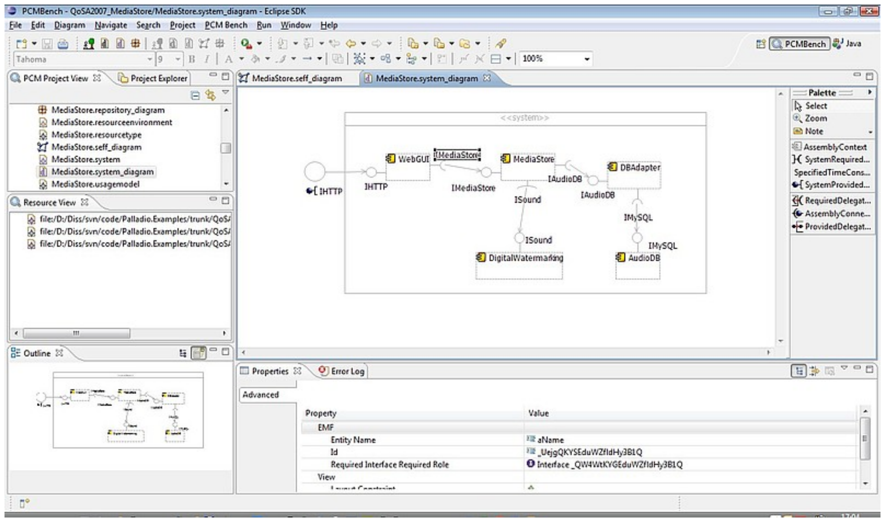
The above-mentioned information about SoftVision looks promising, but there is no example of a detailed view anywhere in the article mentioned; instead a number of abstract views are presented. This brings us to believe that SoftVision was supposed to analyze relations in software architectures instead 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 couldn't fulfill our goals, because it was too general. It also allowed the visualization software architectures, networks and web structures, so it wouldn't be able to initialize for a concrete component model.

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.

4.2 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, like, e.g., SaveCCM [18]). This results in the need for every developer to learn this notation in order to use it and this approach complicates the exchange of diagrams between different domain experts.



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

Figure 7: Palladio – system editor [40]

The next example, which will not be discussed in detail, is a pack of visualization tools for the Palladio component model [2]. 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 7, it doesn't provide much detail, but thanks to other tools that support the development process of Palladio components[6], it is a valuable supplement.

Visualization approaches from this category are able to describe applications of only one component model and they are thus not usable for general visualization. However, it is an interesting example of what is possible when visualization has to support only one component model.

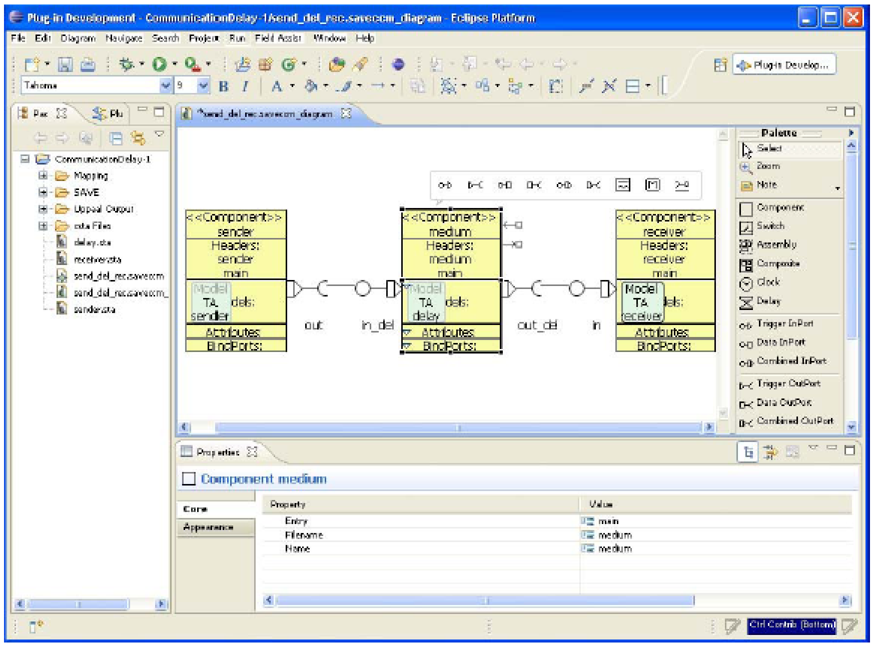


Figure 8: SaveIDE – architecture editor [40]

4.2.1 SaveCCM visualization

The authors of SaveCCM [18] 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 [41] and [40]. 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 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, through behavior testing, by using timed automata.

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

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 8 and 9.

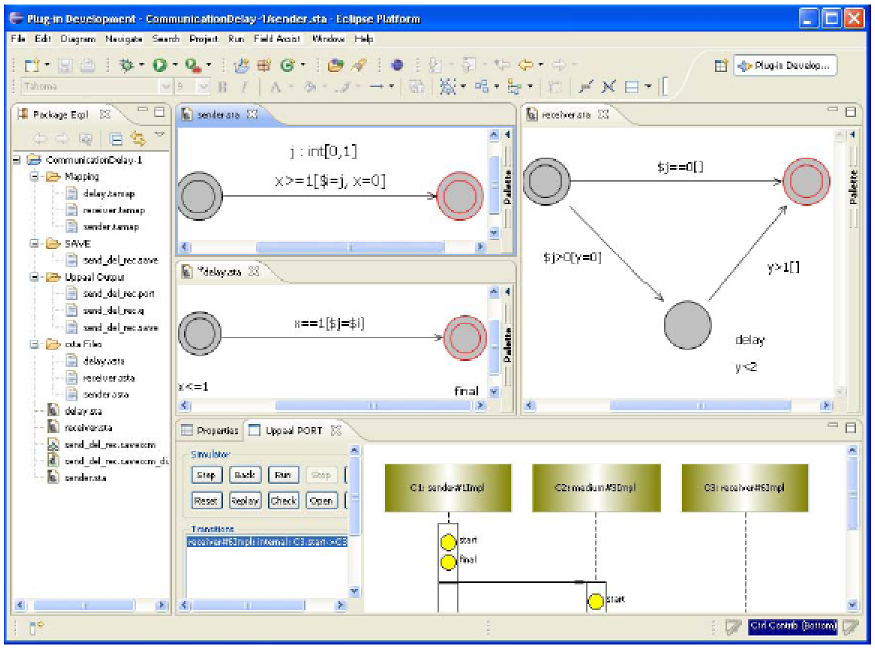


Figure 9: SaveIDE – behavioral editor [40]

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 [36]: just to overcome the difficulties of specific visual notation.

4.3 Summary of current visualization approaches

It might look as if this state of the art is incomplete; however, we did extensive research in the field of different visualization tools and approaches. We found out 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 vanilla 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, which was described in subsection 4.1.4.

A general UML component diagram can be visualized in any UML 2 editor: e.g., MagicDraw[7], Papyrus[8], StarUML[9] and IBM Rational Software Modeler [10]. A general "boxes-and-arrows" diagram can be visualized easily thanks to vast number of visualization libraries: e.g., yFiles[11], Protege[12], Neoclipse[13] and Jgraph[14]. 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 companies, but a general visualization approach, that could represent details is better and is also the goal of this report. However, it is interesting to compare what it is possible to use when one needs to support only one component model.

The best option is 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.

4.4 Meta-models for the description of component-based applications

Data that can 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

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

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 [16], where he stated that the description of component model is vital.

MOF (Meta Object Facility) [29] provides the required basis for the definition of any 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 10.

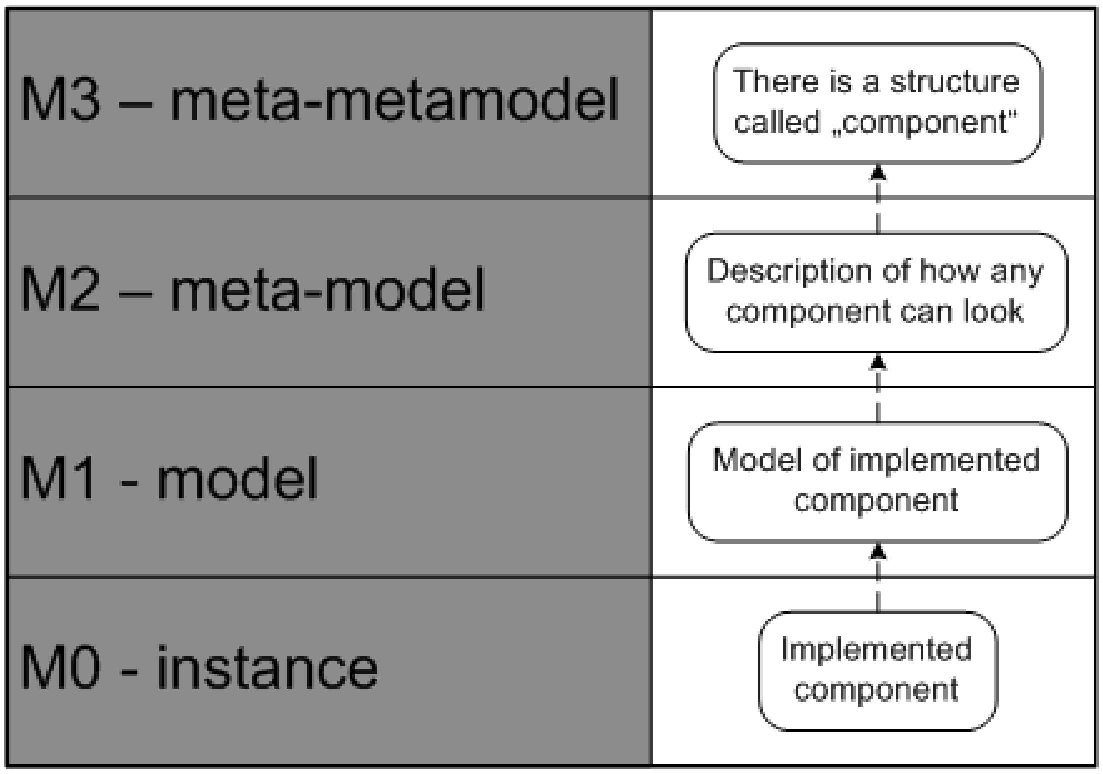

M3 – meta-metamodel
There is a structure called „component"
M2 – meta-model
Description of how any component can look
M1 - model
Model of implemented component
M0 - instance
Implemented component

Figure 10: MOF abstraction levels mapped on the component domain

The implementation of a concrete component that can be assembled and deployed is the lowest level - M0 - which refers to the things from the real world. 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 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. After this brief introduction to MOF philosophy one can say, 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 Subsection 4.1.

In the following subsections we will present the UML meta-model that is able of such description as mentioned earlier. Here we will however focus on concrete ways of extensions provided by the UML meta-model. This is again the only solution available today for such description, but there are also works that are closely related to the description of component applications so we mention them at least as related work.

4.4.1 UML 2 meta-model

UML meta-model [31] is closely bound to MOF (Meta Object Facility) [29] because it is not only defined by MOF, but also is a part of MOF core specification. The UML meta-model defines the "component" classifier from version 2.0 and together with the default extension mechanism in the form of profiles it allows users to define their own component model-specific meta-model. It is important to keep in mind that all information about components are held by the UML model in order to use them in various ways, for instance to visualize them in a UML visualization tool.

UML Profiles offer three types of extension mechanisms:

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 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 [35]. The author used this “heavyweight” extension of UML to provide a better description of C3 architectural style [42].

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.

4.4.2 Related work

In [39] 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 [12] 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 The ENT Meta-Model

The ENT meta-model is a MOF M3 model defining the structures of component models and component-based applications. Previous version [3] of 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 5.4.3 and 5.4.2.

Its main characteristic is the use of the faceted classification approach [38] 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 following subsections, was analyzed in order to create a MOF model. This model was implemented using EMF (Eclipse Modeling Framework) [45]. The process of this implementation is described in [44]

5.1 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 ENT definition described in section 5.3.1. The ENT meta-model enforces this structuring of component interface (as opposed to a flat collection of items, cf. Figure 18) 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, MessageDrivenBeans 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.

5.2 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}
- 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 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 $M = (name,\ C_S)$ where $name \in \mathit{Identifiers}$ 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 5.3.1.2) have traits *Export packages*, *Provided services*, *Import packages*, etc.

Definition A **component type** is a tuple $C^{def} = (name,\ tagset, T)$ where $name \in \mathit{Identifiers}$ is the name of the component type, $tagset = \{tag_i\}$ 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 triples $tag_i = (name_i,\ valset_i,\ d_i)$ where $name_i \in \mathit{Identifiers}$, $valset_i$ is the set of its possible values, and $d_i \in valset_i\ \cup\ \{\epsilon\}$ 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: $\forall C_i, C_j \in M.C_S, i \neq j : C_i \neq C_j \implies C_i.name \neq C_j.name$.

Definition A **trait definition** is a tuple $T^{def} = (name,\ metatype, K, tagset, extent)$ where $name \in \mathit{Identifiers}$ is the trait's name, $metatype \in \mathit{Identifiers}$ is the meta-type of the component interface elements grouped by this trait, K is their ENT classifier, $tagset = \{tag_i\}$ is the finite set of allowed tags of these elements, and $extent \in \{one, many\}$ defines the maximum number of elements in the trait[15].

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

[15]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.

5.3.1 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 [32] plus examples of behavioural and extra-funcional element traits. OSGi was chosen for its industrial relevance, simplicity and ubiquity.

5.3.1.1 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}

5.3.1.2 Trait Definitions For demonstration purposes we provide the definitions of just four traits here, see [51] 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

5.3.1.3 Behaviour and Extra-Functional Properties Traits can also represent other than functional elements, for example a quality of service aspect (e.g. [23]) or the expected call sequence protocol [37]. 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

5.3.1.4 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 11, the architecture should be self-descriptive.

5.4 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

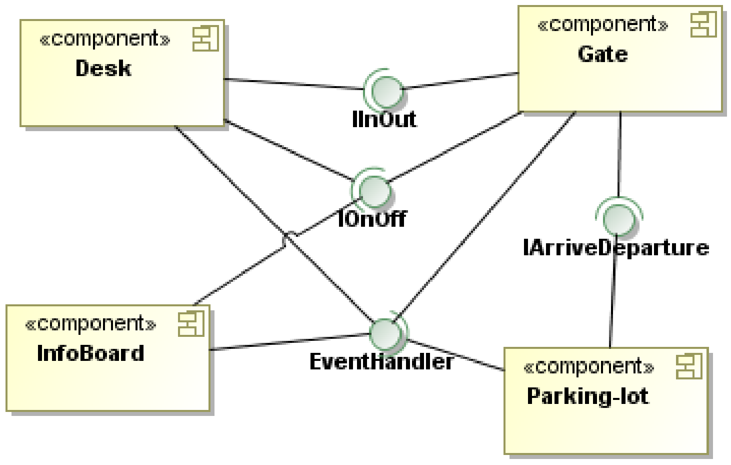


Figure 11: Component application example — Parking Lot (OSGi application)

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.

```
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 12: Manifest file for Gate bundle

5.4.1 Individual Components

In this section an example of the Gate bundle (see Figure 11) 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 12.

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 sub-components and their delegation bindings (see subsection 5.4.3 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 specification 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.

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

Figure 13: The *imported_packages* trait of the *Gate* bundle in ENT representation

Definition An **interface element** e of a concrete component c with specification written in language L is a tuple $e = (\mathit{name}, \mathit{type}, G)$ where $\mathit{name} \in \mathit{Identifiers} \cup \{\epsilon\}$ 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 14: The *parkinglog* element of the *imported_packages* trait in ENT representation

5.4.2 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 15: The service *cz.zcu.kiv.parkinglot.gate* bound to bundle *Desk* in ENT representation

5.4.3 Hierarchical Components

Some component models such as SOFA [7] 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 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 5.4.1): $\forall (e^c, e^s) \in S^d : e^c \in c.t_m, e^s \in s.t_n, t_m.T^{def} = t_n.T^{def}$, 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 11 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.

5.5 Structuring Level: Category sets

Some traits and elements could be at particular times considered as unwanted information when reading 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 display traits.

Definition The **category set** over an ENT model is a pair $Catset = (name, \{(c, K, f)\})$ where $name, c \in \mathit{Identifiers}$ are the names of the category set and its categories, and $f = K \times T^{def} \rightarrow \mathit{boolean}$ 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 5.5 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.

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

Figure 16: The ENT category set

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.

6 Proposed Visualization

In this section we will propose a visualization technique for component-based applications that is based on the ENT meta-model described earlier. This visualization should provide an alternative to the UML component diagrams – describing the structure of any component-based application.

It is aimed to help understand the application faster and more easily and to help in any situation where it is important to keep the scope of the application under control, while having access to the details. The second goal is to remove the necessity of creating multiple diagrams for the same structure, just to differ in the number of details. We decided to address these problems from the reasons mentioned in Introduction.

6.1 Underlying principles

We built our interactive visualization on several principles that help us to achieve these goals. They are adapted from [20] [27], where even more ideas on how to increase cognitive capabilities for information visualization can be found.

First of all, we didn't want to create a new visual notation when it is not needed, so we *reused* several principles of how the components should look and how they should be connected from UML, and added several improvements or novel features that UML does not offer. Another thing that is different 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 6.2.

To eliminate the need for multiple diagrams, we *keep all information stored in one model* and we present only information that is required by the user depending on his role. To enable these requirement-based views, we created Category sets that enable rule-based filtering based on the trait characteristics. Use of category sets is described in Section 6.3.

In diagrams of complex applications, the complexity of connection lines can completely overwhelm users' cognitive capacity. We propose their reduction to only *one connection between two components* and we discuss it in Section 6.4, where we also describe how the details about these connections can be accessed. Similarly, we propose how to *optimize orientation in complex hierarchical applications*, by simply collapsing them to hide details and expanding on demand, thus also working on different levels of details. These principles are discussed in Section 6.5.

The last principle that addresses the needs of component assemblers is "Structure mode". It is designed to help working with the whole structure while not losing all the advantages of our proposed visualization technique. This is briefly described in Section 6.6.

6.2 Visual notation of components

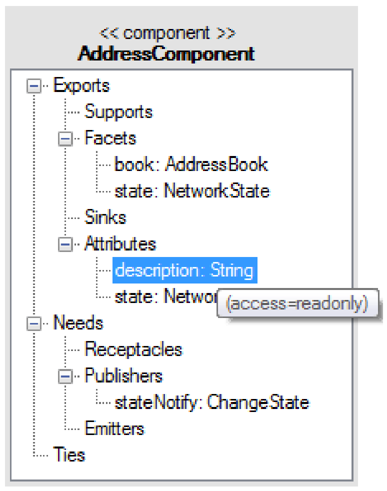


Figure 17: Sample CORBA component in ENT visualization.

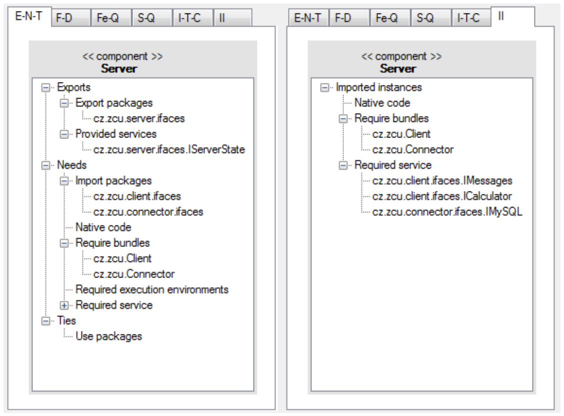
The visual representation of a single component is described here. The header of a component has two lines: the type of component is enclosed by guillemets on the first line and the name of the component is present on the second line. The header and connections between components are the only things that do not change; the body of the component can be altered as the user needs. The component is highlighted when the user clicks on it and all tags related to the component itself are shown in an info box, which appears next to the component.

The body of the component is quite different from UML (see Figure 17). It presents elements in a tree structure. The highest level are categories that group traits which match specified rules (see Section 6.3); the elements are then leaves of this structure.

A single element is displayed in the classical way as *nameOfElement: type*. If it doesn't have any type defined (and also when the type isn't 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 17), unlike for OSGi, where elements are the names of interfaces, classes and packages (e.g. Figure 19).

If the user is interested in a concrete element, he can hover over it and all tags will be displayed in info box. This info box is apparent in Figure 17, where element *description* is *readonly*.

In Figures 17 and 18, it can be seen that *different components from different component models are displayed similarly*, so it is easy to read components from any component model.



```
<< component >>
Server
⊟ Exports
   ⊟ Export packages
      cz.zcu.server.ifaces
   ⊟ Provided services
      cz.zcu.server.ifaces.IServerState
⊟ Needs
   ⊟ Import packages
      cz.zcu.client.ifaces
      cz.zcu.connector.ifaces
   Native code
   ⊟ Require bundles
      cz.zcu.Client
      cz.zcu.Connector
   Required execution environments
   ⊞ Required service
⊟ Ties
   Use packages
```

```
<< component >>
Server
⊟ Imported instances
   Native code
   ⊟ Require bundles
      cz.zcu.Client
      cz.zcu.Connector
   ⊟ Required service
      cz.zcu.client.ifaces.IMessages
      cz.zcu.client.ifaces.ICalculator
      cz.zcu.connector.ifaces.IMySQL
```

Figure 18: Filtering ENT visualization by category sets.

6.3 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.

These problems are solved by using category sets described in Section 5.5. These category sets can filter and group traits and then be used to provide the tree structure described in the previous section.

For example, there are two different views of the same OSGi bundle in Figure 18. The ENT category set shows all traits of the bundle component type, while the second set (II) is very selective and shows only imported instances. The possibilities of grouping and filtering are very rich, as they can use more than one condition.

Additional category sets can be defined by a user and used in the visualization. The visualization of an application can thus be parametrized (modified) to suit individual unforeseen needs or to specific roles. For example, an OSGi system architect would benefit most from the view consisting of two categories for provided and required instances (Construct=Instance) to concentrate on service-based communication.

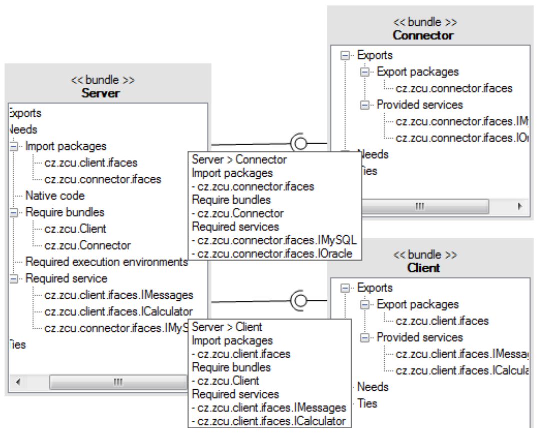


Figure 19: Simple OSGi application in ENT.

6.4 Inter-component bindings

Bindings between two components are represented by a "lollipop" notation. This style was chosen as it is a standard way introduced by UML. In real world component applications, it is usual that there are multiple bindings between two components. With dozens of components this would result in a cluttered diagram. To reduce the complexity of such diagrams, we hide all relations between two components under one line. The user can still study how the components are related together, but the number of connection lines is significantly reduced.

If the user wants to know which elements are creating a connection, he can click on the given line and an information box will appear near the line. In Figure 19, one can see the *Server* bundle that requires several elements from *Connector* and *Client* bundles. In this figure, the user already required information about connections and because of that the info boxes on both connection lines are active.

6.5 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.

Our notation therefore displays composite components similarly to atomic components, without revealing their internal structure, but informing that inner architecture is present by the key word *composite* in the upper right corner of the component.

The user can study a diagram and when he wishes to display how the internal

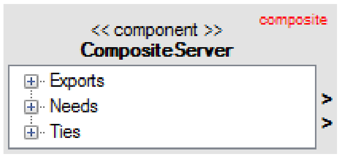


Figure 20: Sample composite component.

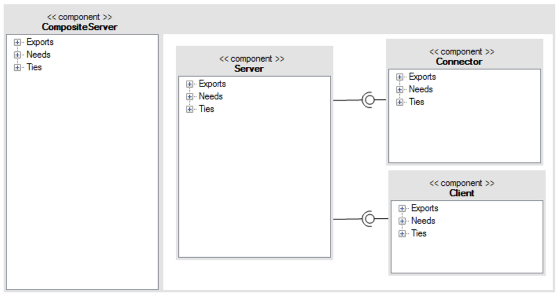


Figure 21: Extended composite component.

architecture of a composite component looks, he just expands its box using the expansion arrows along the edge of the component box to unveil the detailed view (see Figures 20 and 21). This feature also keeps the diagram of the hierarchical application simple and doesn't require the creation of any other separate diagrams to study the structure of composite components.

6.6 Structure mode

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.

Therefore, the structure mode 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. This results in a clean and simple "boxes-and-arrows" diagram. The component sets are still active, so clicking on the component will reveal the body with a selected component set displayed in full detail as usual.

6.7 Comparison with 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.

Table 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

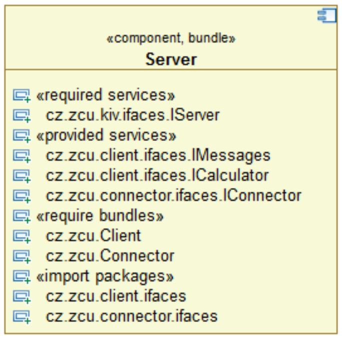


Figure 22: UML extended to look similar to ENT.

The situations in which the two visualization alternatives were compared are presented in Table 1.

This comparison assumes that we use UML with a profile and visual design styled to look similar to our representation, so these two approaches are comparable. We found similar representation in [15], where elements are grouped by their stereotype. The OSGi sample *Server* bundle in Figure 18 can be represented in UML as shown in Figure 22.

7 Future Work

In our research on interactive visualization of component-based applications, we studied a number of different works, which were analyzed to gain experience. Based on this experience, we designed an interactive approach that is built on the ENT meta-model, so it can take advantage of this meta-model. There are features that wouldn't be possible without the ENT meta-model, for example, filtering and grouping of elements in the body of a component. We made a special effort to emphasize interactivity and provide interactive techniques from all categories that were identified in [54] to maximize the value of interactivity, but the fluency of work with the diagram was always kept in mind.

We are currently working on a tool that implements the visualization approach described. This tool is being written as an Eclipse RCP (Rich Client Platform) application. Analyzers of component implementations for OSGi, EJB and SOFA component models are also part of the construction. These loaders are written as plugins and thus support for more component models can easily be added to the finished tool. The completed implementation of this tool will, furthermore, provide extension points where new visualization styles could be added.

The proposed interactive visualization approach has to be tested on realistic case studies and real component applications, involving users (programmers and architects) to provide empirical evaluation. Therefore, these case studies have to be designed, executed with a sufficiently large group of users and analyzed. In the second stage, it may be possible to suggest improvements based on user experience from the case studies. Different kinds of highlighting and coloring based on classification, clustering of components and other enhancements will be the subject of further research to improve the usability of this approach in component-based development.

8 Conclusion

This report suggests that it is important to create a new approach that is able to visualize component-based applications more efficiently than UML does. This approach should implement ideas, which are also part of this report, which emphasize the importance of interaction in the visualization of complex applications.

The proposed visualization approach reuses proved and well known visual notation of UML components, which is the base for new notation that is able to better suit the demands of CBSE. The interaction techniques were chosen after analysis of different interaction categories and study of principles of the human brain and cognitive principles needed to create a mental model of the visualized application. The overall goal of this new visualization approach is to use the maximum potential of the human brain, to boost the process of learning the unknown application in detail.

The structure which is capable of holding information about any component-based application is not trivial and is needed for visualization of this information. This report proposes the ENT meta-model, which is able to describe any component-based application in great detail, based on previous analysis of the concrete component model. This meta-model also offers an added value in the form of content-awareness of stored data, thus these data can be interpreted in another way - it is not bound to our proposed visualization.

The state of the art of current visualization approaches of component-based applications and meta-models, able to describe these applications, was discussed to justify development of both the new meta-model and the visualization approach using this meta-model. This state of the art might look incomplete, but this is due to a lack of research in this closely specialized field of visualization.

Further verification of the proposed visualization approach should be provided to evaluate the correctness of this approach or to find the shortcomings that should be removed. There is also space for future improvements that could help even more in the process of understanding or analyzing. The finalized and well-tested approach should provide sufficient background for a full Ph.D. dissertation.

References

[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] Martin Buchi and Wolfgang Weck. A plea for grey-box components. Technical report, 1997. http://www.ncstrl.org:8900/ncstrl/servlet/search?formname=detail&id=oai

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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*, October 2007.

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

[14] 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.

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

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

[17] 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.

[18] 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.

[19] 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.

[20] Ric Holt. Software Architecture as a Shared Mental Model, 2002.

[21] 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.

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

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

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

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

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

[27] 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.

[28] Object Management Group. CORBA Components, 2006.

[29] Object Management Group. Meta Object Facility (MOF) Core Specification, 2006.

[30] Object Management Group. UML Profile for CORBA and CORBA Components Specification, 2007.

[31] Object Management Group. UML Superstructure Specification, 2009.

[32] OSGi Alliance. OSGi Servise Platform Core Specification, 2009.

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

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

[35] 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.

[36] 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.

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

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

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

[40] 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.

[41] 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.

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

[43] Matti Sillanp and Alexandru Telea. Demonstration of the softvision software visualization framework, 2005.

[44] 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.

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

[46] Sun Microsystems, Inc. Enterprise JavaBeans(TM) Specification, 2001.

[47] Sun Microsystems, Inc. Enterprise JavaBeans(TM) Specification, Version 3.0, May 2001.

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

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

[50] 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.

[51] 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.

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

[53] 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.

[54] 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 Trans. Vis. Comput. Graph*, 13(6):1224–1231, 2007.