Západočeská univerzita v Plzni

Fakulta aplikovaných věd

VIZUALIZACE ROZSÁHLÝCH DIAGRAMŮ KOMPONENT

Ing. Lukáš Holý

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

Školitel: Doc. Ing. Přemysl Brada, MSc. Ph.D.
Katedra: Katedra informatiky a výpočetní techniky

Plzeň, 2014

University of West Bohemia

Faculty of Applied Sciences

LARGE COMPONENT DIAGRAMS VISUALIZATION

Ing. Lukáš Holý

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

Supervisor: Doc. Ing. Přemysl Brada, MSc. Ph.D.
Department: Department of Computer Science and Engineering

Pilsen, 2014

Prohlášení

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

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

V Plzni dne ..... Ing. Lukáš Holý

Abstrakt

Softwarové aplikace se dnes mohou jednoduše skládat ze stovek nebo i tisíců komponent a je proto složité porozumět jejich struktuře. Zobrazení diagramu příliš situaci nepomáhá, jelikož ten většinou obsahuje vizuální šum způsobený velkým množstvím komponent a jejich spojení. To platí zejména pro ploché (nehierarchické) komponentové modely.

Tato práce shrnuje současný stav poznání v oblasti nástrojů a přístupů k vizualizaci komponentových diagramů a ukazuje, proč tato oblast stále obsahuje témata k výzkumu. Následně navrhuje sadu kritérií pro zhodnocení nástrojů pro vizualizaci komponentových diagramů.

Jako odpověď na identifikované potřeby a výzvy představujeme nový přístup k vizualizaci, který zjednodušuje orientaci a navigaci ve složitých diagramech. Ten je mimo jiné užitečný v procesu reverzního inženýrství. Jedním z klíčových konceptů tohoto přístupu je odstraňování velkého množství spojení z diagramu beze ztráty informace o propojení.

Dalším konceptem je technika zvaná viewport, která je taktéž použitelná v UML diagramech komponent. Tato technika zjednodušuje práci s komplexními diagramy zvýrazňováním detailů důležitých částí diagramu a jejich okolí beze ztráty celkového přehledu. Část naší práce se také zaměřuje na vizualizaci mimofunkčních charakteristik v komponentových diagramech.

Abychom byli schopni prokázat, že navržené techniky ulehčují práci, implementovali jsme webový nástroj nazvaný CoCAEx. Provedli jsme zhodnocení nástroje formou porovnání časů jednotlivých úkolů v nástroji CoCAEx a jiném, běžně v průmyslu používaném, nástroji. Z této studie vyplývá, že CoCAEx pomáhá urychlit proces reverzního inženýrství.

Abstract

Software applications can easily consist of hundreds or thousands of components and it is thus difficult to understand their structure. Diagram visualization does not help much because of visual clutter caused by big amount of elements and connections, especially in the case of flat component models.

This thesis sums up current state of the art tools and approaches in component diagrams visualization and shows why cannot cope with the challenges brought by diagrams of large component systems. After that we propose a set of criteria for the evaluation of tools for component architecture visualization.

As an answer to the identified needs and challenges we present a novel approach which eases the orientation and navigation in complex diagrams. It is among other benefits useful in the reverse engineering process. One of the key concepts of this approach is removing a large part of connections from the diagram while preserving the information about component interconnections. Another one is the viewport technique for use in the visualization of UML component diagrams. This technique eases the work with complex diagrams by highlighting details of the important parts of the diagram and their related surroundings without losing the global perspective. Part of our work also focuses on extra-functional properties visualization in component diagrams.

To be able to prove that techniques proposed in the thesis ease the work with large component diagrams, we implemented them in the web-based tool called CoCAEx. We performed the evaluation where we compared time of tasks in CoCAEx and other commonly used industrial tool. It shows that CoCAEx helps speed-up the reverse engineering process.

Resumo

Actualmente, as aplicações de software podem ser compostas por centenas ou milhares de componentes, e por essa razão é difícil entender a sua estrutura. A visualização de diagramas não é muito útil porque habitualmente os mesmos contêm ruído visual causado por uma grande quantidade de componentes e ligações entre os mesmos. É nomeadamente o caso de modelos planos de componente (não hierárquicos).

Este trabalho de Doutoramento resume o conhecimento existente na área de ferramentas e abordagens para a visualização de diagramas de componentes, evidenciando que nesta área ainda há espaço para investigação. Subsequentemente, o trabalho propõe um conjunto de critérios para avaliar ferramentas de visualização de componentes.

Como a resposta para as necessidades e desafios identificados apresentamos uma nova abordagem à visualização de componentes que facilita a orientação e navegação em diagramas complexos. Por outro lado, a abordagem proposta é também útil no processo da engenharia reversa. Um dos maiores contributos da abordagem é a remoção de muitas ligações dos diagramas, sem que seja perdida informação relativa a interligações.

Outro contributo desta tese é a técnica denominada de viewport, aplicada em diagramas UML de componentes. Esta técnica simplifica o trabalho com diagramas complexos porque realça os detalhes de partes importantes do diagrama e as suas proximidades, sem que se seja perdida a orientação geral. Uma parte do nosso trabalho põe também em foco a visualização de características não-funcionais em diagramas de componentes.

Para demonstrar que as técnicas propostas facilitam a compreensão, foi desenvolvida uma ferramenta web chamada CoCAEx. Avaliámos a ferramenta através da comparação dos tempos de tarefas individuais utilizando a ferramenta CoCAEx versus outra ferramenta vulgarmente utilizada na indústria. Conclui-se deste estudo que a CoCAEx ajuda a acelerar o processo de engenharia reversa.

Contents

List of Figures

Chapter 1

Introduction

This work focuses on the effective visualization of large system component diagrams. New methods of the visualization should bring clarity of represented data and speed up a process of understanding of unknown applications. These methods should support user interaction with the diagram for better customization according to user needs.

1.1 Introduction to Component Software Modelling

Software architects and developers have been using various forms of visualizing the structure of software applications since the advent of the discipline. In the last 20 years, the increased adoption of object-oriented programming lead first to several proposals for adequate modelling notations which were then gradually consolidated into the current standard – the Unified modelling Language (UML) [85]. While UML is able to model both the static and dynamic aspects of many kinds of software, recent development in the field of component-based software engineering (CBSE) brings new challenges regarding UML usability [80], [82].

The visualization of component-based applications [103] is not a trivial task due to the rich structures of component interfaces and the differences between component models. Frameworks like EJB [101], CORBA [84], OSGi [86] and more can be found in commercial applications and even more component models – for example SOFA [23], Fractal [78] or CoSi [21] – are the subject of research.

The diversity of component models in terms of the features available on component interface is well described in e.g. [31]. On an abstract level, components have in common two basic properties: the black-box nature and the fact that the features they need and provide on their interface are well defined [103].

Their interface features can cover all known contract levels according to [18]:

- syntactic, e.g. functional interfaces in most models and events in EJB3 [101],
- semantic, e.g. triggers in SaveCCM [44],
- behavioural like protocol in SOFA [87],
- extra-functional property specifications, e.g. in Palladio [15],
- control interfaces like in Fractal [78].

We can also refer to a different classification according to [32] as stated in Section 2.3. Above mentioned richness indicates that modelling and visualizing component applications is a challenging task.

1.2 Problem Definition: Diagram Complexity

Software applications become more and more complex [106]. Although there are lots of tools that help the development and reverse engineering process, they are still limited in helping human understanding of the application structure and substantial research is still needed [26].

As stated in [49]:

> UML class diagrams furthermore suffer from scalability problems and should therefore only be used for small/partial software systems, e.g., a dozen interrelated classes. ... Straightforward visualizations of a software system hierarchy as well as its function call graph suffer from visual clutter.

Software components [103] are one of the ways to handle this complexity as they encapsulate parts of functionality to unified components. Even with the usage of the components, nowadays applications can easily consist of hundreds or thousands of them. It is therefore difficult to explore the structure of the application and create a mental model of the whole system.

One of the ways how to get an insight into a component application structure can be a diagram, e.g. UML component diagram. When the diagram is large there are many problems with exploring it. They come from the contradictory need of providing enough details and showing the complete diagram (application structure) at the same time.

The main problems are following:

- Diagrams displayed at the desired level of detail become **too big to keep orientation** and fit on reasonable displays.

- It is difficult to **trace dependencies between distant components**, as shown in Figure 1.1.
- When displaying the whole diagram on standard screens, individual elements are hard to recognize and often there is visual clutter caused by dependency visualization. Thus next question is **how to reduce visual clutter [92] caused by the large number of elements and connections between them**. The visual clutter makes tracing of dependencies difficult and hinders orientation in the diagram, as shown in Figure 1.2.

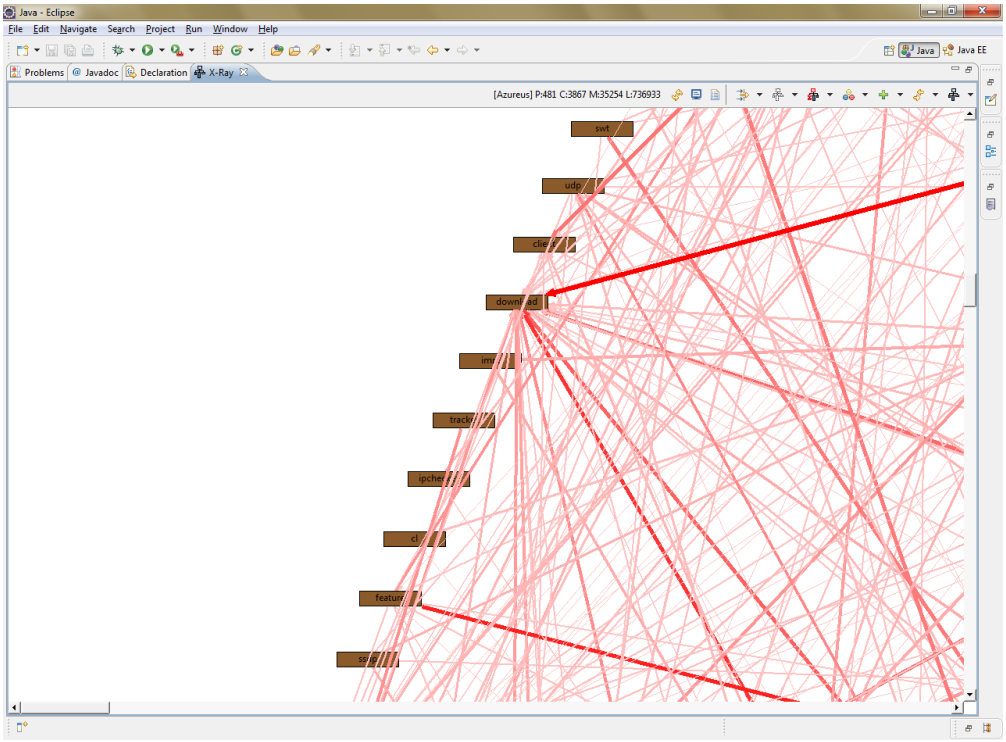


Figure 1.1: Higher Level of Detail in Complex Diagram (Azureus Application in X-Ray Tool)

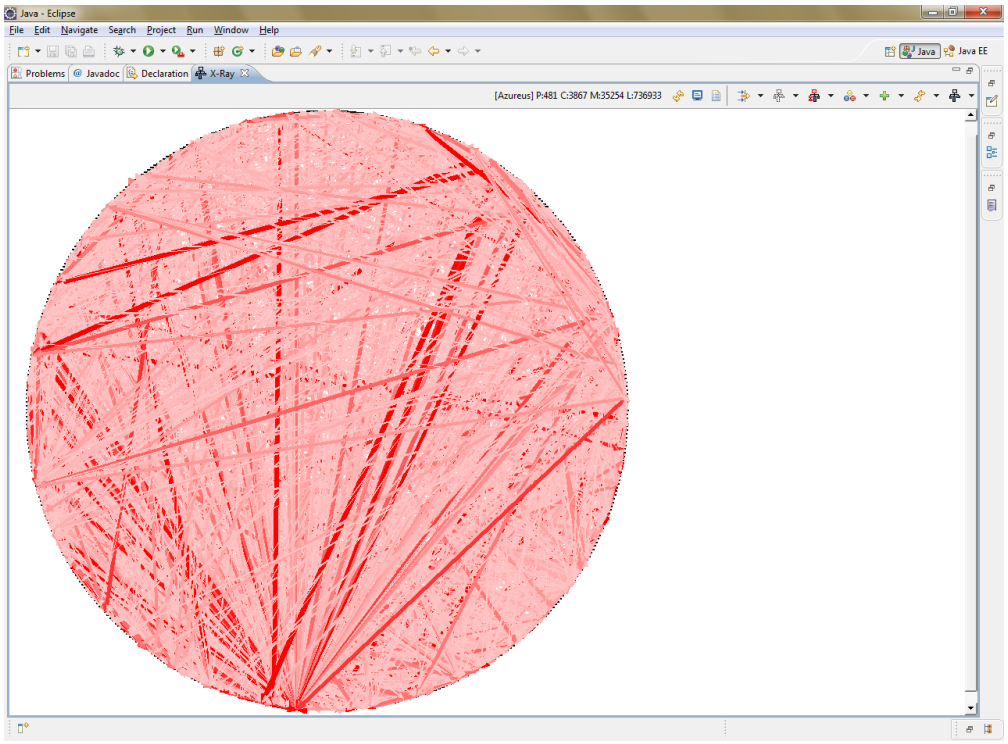


Figure 1.2: Overview of Complex Diagram (Azureus Application in X-Ray Tool)

Although tool support helps reducing time and cost [100], current tools do not offer features designed for work with such large diagrams effectively [58].

It is possible to divide large diagram into smaller ones. But in this case user would lose the overview of the whole system and the information about interconnections among system parts. Although diagrams of hierarchical component models [23] usually does not have this problem because they keep the information about parts in their hierarchy, there are lots of component models [101], [86] with flat structure where the described problem occurs. There are also other software and non-software structures (data models, network architectures, transport systems, power grids, ...) where we can identify similar problems.

1.3 Goal of the Work

The main goal of the thesis is to:

> Bring better and more effective ways of large component software visualization to reduce the time needed to understand the application structure.

To fulfil the goal we have to design a new visualization approach, thus we state two sub-goals and one approach:

- sub-goal: Our main focus will be on techniques that will **not modify the visual representation of UML component diagram rapidly**

to shorten the learning curve of potential users. This will allow the techniques as well as the implementation to be adopted easier, which has been identified as a problem of current tools in [26]. It also means that we will mainly focus on node-link diagram representation. Node-link diagrams are widely used in various domains and thus inventing new techniques in software visualization can be potentially generalized for graphs used in other domains.

- sub-goal: **visualize a sufficient amount of detail to be able to exactly identify individual components**.
- approach: provide ways to **hide less important details and show them on demand**.

We will provide an evaluation of discovered techniques. For that purpose we will provide an implementation of invented techniques.

1.4 Structure of the Thesis

The work is structured as follows. Chapter 2 provides overview of the component based software development. Chapter 3 describes topics relevant to software and graph visualization. Chapter 4 shows evaluation of existing approaches in component software visualization. Chapter 5 describes the our approach to large component diagrams visualization, which is called Complex Component Applications EXploration (CoCAEx). For the ability to evaluate our approach we implemented it in research tool. It is described in Chapter 6. For the evaluation purposes we performed a user study. Chapter 7 presents its results. Finally, Chapter 8 concludes the paper with the summary of findings. It also provides evaluation of thesis goals and future work.

Chapter 2

Component Based Software Development

Component based software development (CBSE) should speed up the development of new software by reusing the existing components. These can be developed by third parties. It should also increase the predictability of produced application. On the other hand there is some overhead in wrapping functionality into components. User of third party components should check the changes of used versions. There is also diversity in component models and frameworks, which slows down the growth of large market. To clarify the term component, its definitions follow.

Szyperski defined components in [103] as following:

> 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.

Bachmann [14] states:

> 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.

Component is [14]:

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

From the point of view of this work, it is important that the diagrams of component-based applications are used in the component based software development. These can be used at the initial phase of designing or composing the application. Furthermore, we can speed up the understanding of an application while doing a change in an existing one. The latter is usually a reverse engineering, an extensions creation or a functionality change process. We provide more details about this topic in Chapters 3, 4.

2.1 Component Models and Frameworks

Bachmann [14] explains terms component model and framework as:

> The component model gives uniformity to components and their composition. Its use is to define how a component should look like, how components communicate each other, which resources they use, etc. The component model ensures the components are compatible in terms of deployment, the communication, etc. It determines the rules components must hold to be able to cooperate and it minimalists misunderstood assumptions. A component framework is basically an implementation of a component model. It supports all mechanisms such as deployment, synchronization, life-cycle, communication of components which are defined in the component model.

Component models will impose standards and conventions of the following kind [14]:

- component types,
- interaction schemes,
- resource binding.

2.2 Compositional Forms

There are several ways of composing components to systems. The possible compositional forms influence the design decisions while designing new component diagram visualization approach. The following subsections present the compositional forms, adopting the classification developed by [14].

2.2.1 Component Deployment

Components (as marked with C in Fig. 2.1) must be deployed into frameworks (as marked with F in Fig. 2.1) before they can be composed or executed. The deployment contract(s) (as shown at point 1 in Figure 2.1) describes the

interface that components must implement so that the framework can manage their resources.

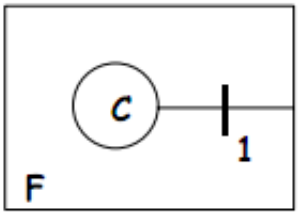


Figure 2.1: Component Deployment [14]

2.2.2 Framework Deployment

Frameworks may be deployed into other frameworks (as marked by F1 and F2 in Fig. 2.2). Contract is analogous to the component deployment contract.

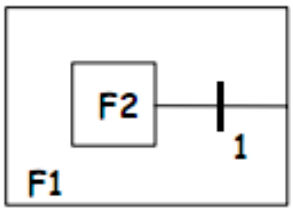


Figure 2.2: Framework Deployment [14]

2.2.3 Simple Composition

Components deployed in the same framework can be composed (as shown in Fig. 2.3). The composition contract (as shown by mark 1 in Fig. 2.3) expresses component- and application-specific functionality; the interaction mechanisms to support this contract are provided by the framework.

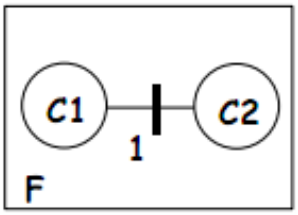


Figure 2.3: Simple Composition [14]

2.2.4 Heterogeneous Composition

Support for tiered frameworks implies composition of components across frameworks, whether across hierarchical (as illustrated in Figure 2.4) or peer frame-

works. In either case bridging contracts are needed in addition to composition contracts (as shown at point 2 in Figure 2.4) in order for interactions to span generic component models.

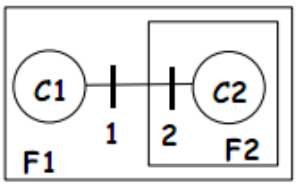


Figure 2.4: Heterogeneous Composition [14]

2.2.5 Framework Extension (Plug-In)

Frameworks may be treated as components, and may be composed with other components. This form of composition most commonly allows parameterization of framework behaviour via "plug-ins." Standard plug-in contracts for service providers are increasingly common in commercial framework.

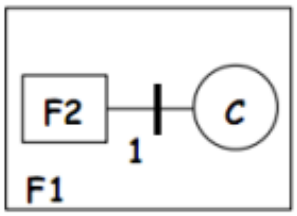


Figure 2.5: Framework Extension (Plug-In) [14]

2.2.6 Component (Sub)Assembly

A component-based system is an assembly of components. The ability to predict the properties of assemblies suggests a similar ability for subassemblies. Contract is used to compose C1 and subassembly C3, which contains one or more components. A question that arises is whether C2 is visible outside of C3 and whether it is separately deployed.

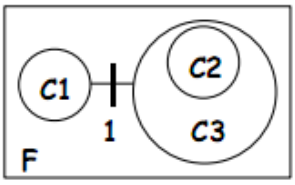


Figure 2.6: Component (Sub)Assembly [14]

2.2.7 Compositional Forms Examples

Most common compositional forms are component deployment and simple composition, which we can find for example in OSGi[1]. Component (sub)assembly is the form represented in hierarchical component models such as SOFA 2[2]. The idea of a framework deployment form can be found for example in SpringDM[3] deployed in OSGi. In this situation we can find the idea of heterogenous composition between SpringDM and OSGi components. It is also possible to extend SpringDM by components running in the OSGi framework (eg. Equinox[4]).

For purposes of this work we limit us to visualize the interfaces as defined in UML component diagram.

2.3 Contracts

As mentioned in previous sections, for communication among components interfaces are used. There are several languages for interface description according to [32]:

- modelling languages (such as UML or different ADLs),
- particular specification languages (Interface Definition Languages),
- programming languages (such as interfaces in Java),
- some additions built directly in a programming language.

There can be also different types of interaction [32]:

- port-based where ports are the channels for communication of different data types and events;
- functions in programming languages defining input and output parameters;
- interfaces or classes in Object Oriented programming languages.

The interfaces provide most of time a basic description of services and thus there are contracts for better description. Contracts among components should guarantee good interface connecting and determine "rights and duties" of components involved. Contracts can be negotiated by involved sides and can be also changed in runtime, if all sides agree. They can also expire.

Contract definition according to [103]:

[1] http://www.osgi.org/
[2] http://sofa.ow2.org/
[3] http://www.springsource.org/osgi/
[4] http://www.eclipse.org/equinox/

> A contract (an interface together with its specification) mediates between independently evolving clients and providers of the services the interface makes accessible.

Beside levels of contracts mentioned in Introduction, we can also state following categorization according to [19]:

- **Syntactic** (or basic) The goal is to make the system work. It is generally specified with Interface Definition Languages (IDLs), as well as typed object-based or object-oriented languages. It ensures the components can be assembled.
- **Behavioral** The goal is to specify each operation. It is generally specified with a couple of assertions: a precondition and a postcondition. It ensures the operations offered and required are not only syntactically compatible but also semantically.
- **Synchronization** The goal is to specify the coordination of operations. It can be specified with an automaton labelled with operations. It ensures the operations are used in the proper order.
- **Quality of Service** The goal is to quantify a few features associated to operations. Performance, availability and quality of result can be specified and negotiated at that level.

Bachmann [14] distinguishes between *component contracts* and *interaction contracts* and defines them as:

- A **component contract** specifies a pattern of interaction rooted on that component. The contract specifies the services provided by a component and the obligations of clients and the environment needed by a component to provide these services.
- An **interaction contract** specifies a pattern of interaction among different roles, and the reciprocal obligations of components that fill these roles.

2.4 Extra-functional Properties

Following section was published in [64].

Despite partial success of CBSE, several issues remain unsolved. One of the important ones Extra-functional properties (EFPs) in CBSE are believed to improve compatibility verifications of the components.

EFPs systems have been addressed from several directions and a lot of approaches have been proposed. For instance, one of the groups proposes independent descriptions of EFPs [36, 12, 72, 42, 79]. While this group splits the

EFP description from their application, different group concerns modelling of the EFPs as a part of a software design [70]. These groups treat EFPs rather independently while a group of comprehensive component models exists taking EFPs natively into account [16, 95].

On the one hand, these component models are often still under research. On the other hand, practically used industrial models such as OSGi or Spring only slowly adopt systematic EFP support. One of the reasons may be wide misunderstanding of what EFPs are [41] that eventually leads to only a partial and non-systematic EFPs adoption in practise. As a suggested solution, general mechanism consolidating EFPs understanding among different vendors as well as different applications has been proposed in [62].

For purpose of this thesis we use EFP definition from [64]:

> An extra-functional property holds any information, explicitly provided with a software system, to detail characteristics of the system apart from the system genuine function to enrich clients with understanding or usage of the system supported by technical [computational] means.

From above mentioned description, we can see that visualizing EFP can be challenging task due to various possibilities of its particular form. We describe this topic more in Sections 5.8 and 6.1.6.

Chapter 3

Software and Graph Visualization

Visualization is very effective way of understanding software structure, behaviour or evolution. This section describes related software visualization approaches and techniques, which help increase understanding the software.

For the purpose of this work, we adopt the following definition of software visualization [109]:

> 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.

In this thesis, we will mainly focus on node-link graphs. Basically we can see several visualization elements in such graph visualization such as nodes, edges and background. Moreover we can apply a graph layout or clustering to help a user identify related elements faster. On top of that, we can choose one of the information schemes which will determine the orientation and navigation techniques. When designing new visualization, it is also important to follow known guidelines and best practices to support mental model creation. All of above mentioned topics are covered by respective sections in this Chapter.

We will also focus on the problems, which arise from large sofware visualization. There is usually large number of various elements in the visualized system and the visualization is cluttered. It is thus suitable to use techniques for visual clutter reduction beside standard visualization techniques.

Visual clutter can be reduced by many techniques, such as bundling (see 3.5.1), sampling [89], clustering (see 3.7) etc. The position of these techniques in the general taxonomy has been described by Ellis and Dix in [34], we describe relevant techniques from this taxonomy in above mentioned sections.

3.1 Mental Model Creation

While doing a reverse engineering, a user creates a mental model of a system from the perception, imagination and comprehension of discourse. Mental model creation is described deeper in [65].

Specifically to software architectures can be the theory of mental models applied as described in [48]. He mentions several laws where following are related to our work:

- **Law of maximal ignorance.** *Don't learn more than you need to get the job done.*
- **Cognitive miser principle.** *Don't waste brain power.*
- **Aesthetic principle.** *Visualization leads to cleanliness.*

Large component diagrams contain lots of elements and visual clutter. Above mentioned priciples lead us to the approach of diagram simplification. It can be achieved by hiding details so that a user does not need to waste the brain power or learn more than actually needed. Hidden details can be shown interactively on demand. The aesthetic principle can be supported by using suitable layout that will show a diagram which is easier to explore.

3.2 Information Schemes

When visualizing complex structures we usually face the problem of not having enough space on the screen to visualize the whole diagram in the desired level of details. Thus we are forced to use some technique to navigate through such a large diagram while showing only part of it on the screen. There are several main approaches while dealing with the complexity problem [30]:

- overview and detail,
- pan and zoom,
- focus and context.

These principles can be combined together to offer a user good understanding of large diagram.

3.2.1 Overview and Detail

This approach (illustrated in Figure 3.1) is very commonly used in the software diagram tools as well as other visualizing fields like maps, CAD systems etc. Its main principle is to provide user two or more views with different level of details. Most common is using the detailed view for most of the screen area

while the overview area is smaller for ensuring orientation. This approach is useful in large diagrams, but its scalability for very large diagrams is limited. It can be partly improved by using more overview levels, but it decreases the transparency of the whole approach.

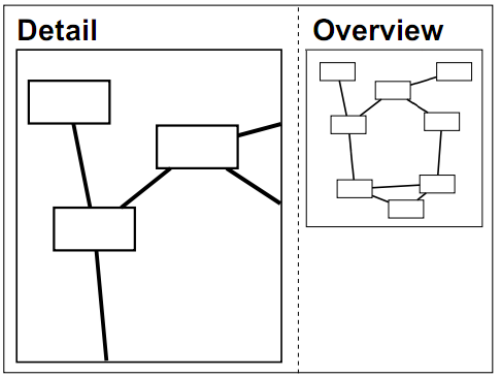


Figure 3.1: Overview and Detail Example

3.2.2 Pan and Zoom

This approach is used for providing the ability to view a desired part of the diagram in desired level of detail. The panning feature usually moves the undelaying diagram according to mouse movements. The zooming feature provides the ability to see the diagram in different levels of size and detail. It is usually handled by mouse wheel, plus and minus keys or buttons dedicated for mouse control. This approach shows the focused and contextual information in views, which are in fact separated by time.

3.2.3 Focus and Context

This approach combines the both focus and context information into one view, as illustrated in Figure 3.2. Focused area shows detailed information, the context area shows the relevant contextual information and they are seamlessly integrated into one view. This integration can be achieved by several techniques such as fisheye distortion, using the border for various types of marks or showing proxy elements for hidden objects. This approach differs from the overview and detail in showing the detail view right in the diagram where the less detailed information is shown. In the contextual part can be also shown information which is out of the focused area. This information cannot be easily shown by overview and detail or pan and zoom approaches.

Figure 3.2: Focus and Context Example [61]

3.2.4 Animation

Animating the changes between showing different views helps a user to better understand a diagram that is shown. It can be used for various changes such as changing zoom level [17] [107], moving between distant nodes in the diagrams or moving the elements during diagram modifications (e.g., layout changes). Important factor while using animation is the time an animation takes. Longer time leads to better understanding of content, but it can slow down work with a tool. Appropriate values for the animation are suggested between 300 and 1000 milliseconds in [67]. Also the work of [108] about optical flow reduction can be helpful while designing an animation technique.

3.3 Graph Layouts

The node layout of a visualized diagram can significantly increase understanding of the application. There are many methods for graph layouts creation such as:

- force-directed,
- orthogonal,
- circular,
- tree,
- layered.

Above mentioned layout methods are briefly described in following paragraphs that describe their relevance for this thesis.

This overview of layout techniques/algorithms is based on a survey of a rather disparate set of sources, from technical documentation to dedicated monographs [33], [2], [28], [47], [88], [43].

3.3.1 Force-directed Layouts

For component diagrams visualization are suitable force-directed graph-drawing methods, as illustrated in Figure 3.3. In these methods the nodes layout is computed according to underlying physical model. The iterative algorithm computes the nodes' placements until the energy in the whole system is minimal.

Classical force-directed algorithms like [38], [66] are suitable for drawing general graphs. They are also used in practice [22] for graphs containing hundreds of vertices. There are also available more efficient force-directed techniques for even larger graphs (tens of thousands of nodes), such as [45], [110].

Figure 3.3: Force- Directed Layout Example [46]

3.3.2 Orthogonal Layouts

Orthogonal methods are using only horizontal and vertical directions for drawing the edges, as illustrated in Figure 3.4. These layouts are used in domains that have orthogonal constraints. Using such layout for large component diagram is not suitable, because it can be tedious to trace dependencies in such one while having a detailed view.

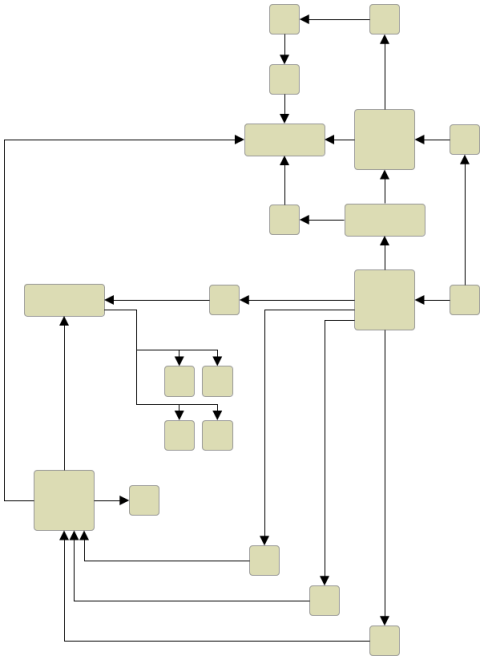


Figure 3.4: Orthogonal Layout Example [11]

3.3.3 Circular Layouts

Circular layouts place the nodes on the circle and the edges connect them inside or outside a circle, as illustrated in Figure 3.5. The edges can be drawn straight (inside a circle) or bended (both inside and outside). Also edge bundling techniques (see Section 3.5.1) are suitable to be used for this layout. Nodes placements on a particular position on the circle can be optimized to minimize the edge crossings.

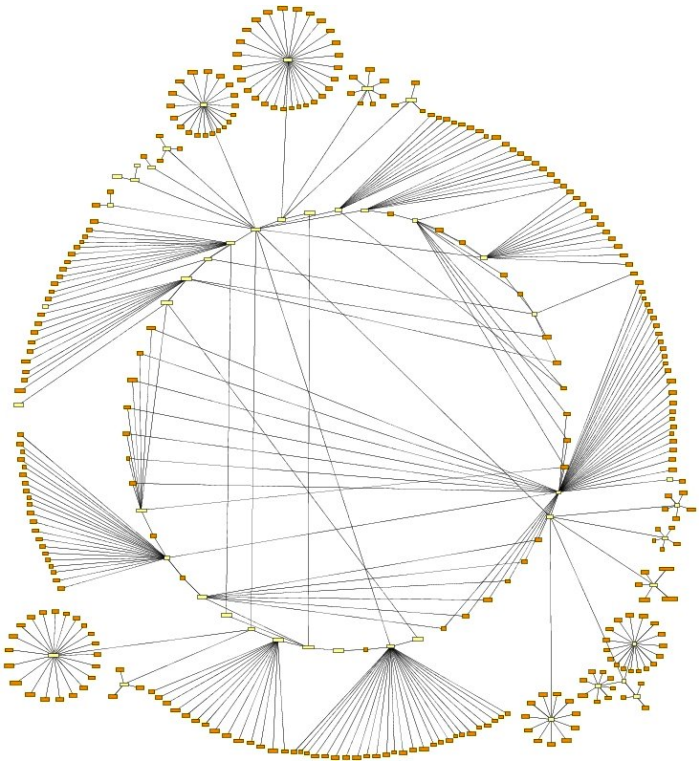


Figure 3.5: Circular Layout Example [11]

3.3.4 Tree Layouts

Tree layouts are suitable for drawing tree graphs, as illustrated in Figure 3.6. Usually the root of a tree is drawn in the middle and its children are placed around it. The component diagrams are mostly not having a tree structure.

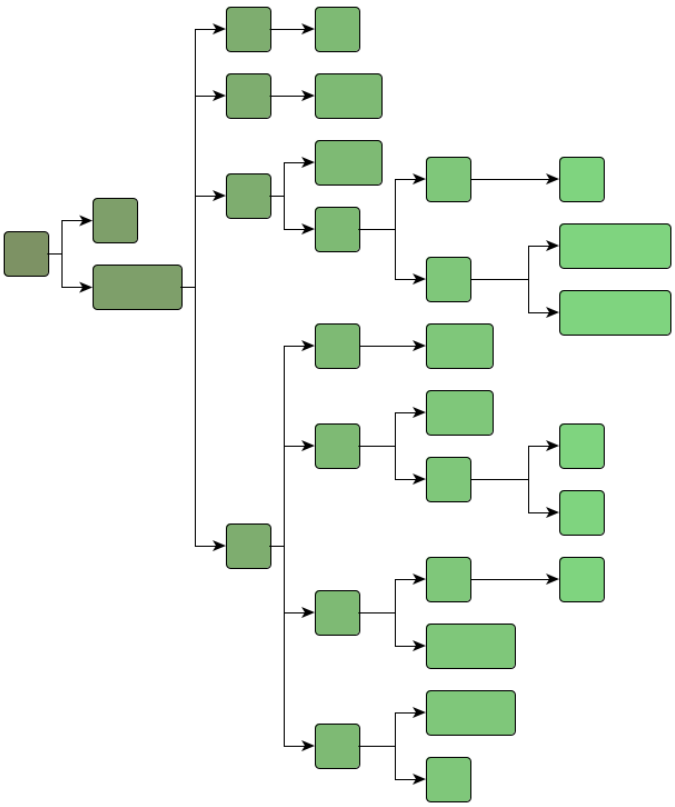


Figure 3.6: Tree Layout Example [11]

3.3.5 Layered Layouts

Layered layouts are suitable for acyclic or nearly acyclic graphs, as illustrated in Figure 3.7. They place the nodes into layers (usually horizontal). Layers are connected among each other and the nodes in each layer are placed to minimize lines crossings among layers. Applications can have a layered architecture, so these layouts can be suitable for their visualization. On the other hand, relying on a fact that unknown application uses such architecture can lead to problems with its diagram exploration, in case the application is in fact not layered.

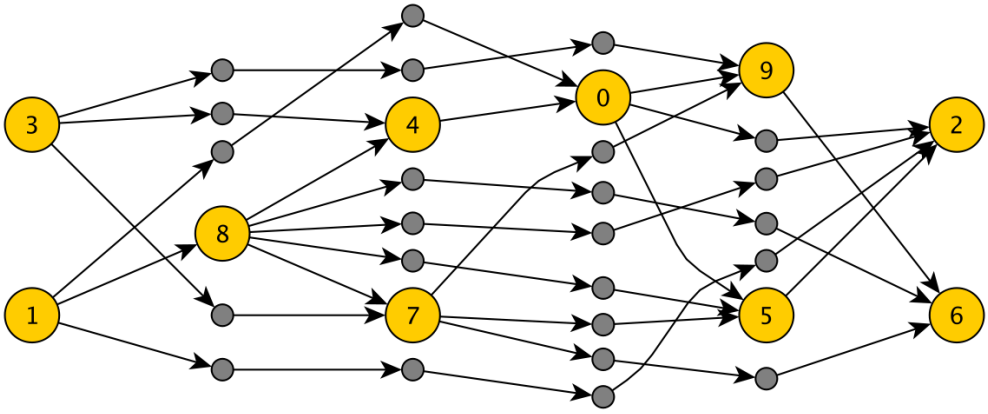


Figure 3.7: Layered Layout Example [6]

3.4 Nodes Visualization

Nodes represent individual software elements, e.g. components. From the visualization point of view there are several main node factors, which can be adjusted to express desired metrics or attributes:

- dimensions - such as width, height (or depth in 3D),
- colors - including various colour effects (e.g., trasitions, patterns),
- shape,
- time validity (when showing changes),
- animations - for showing both static and dynamic information.

While using various node dimensions and colors can be in accord with visual syntax of used model, changing the shape of the node is usually violating it.

The work of Anslow [27] uses basic shapes combined with colors to represent individual nodes as shown in Figure 3.8.

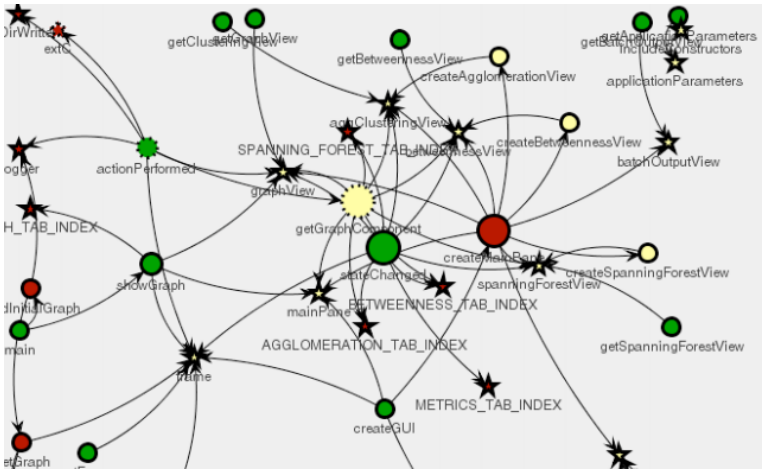


Figure 3.8: The ExtC Graph View using various node representations [27]

The work of Sazzadul [13] shows the application in 3D as a city, where are various buildings used for node representation, as shown in Figure 3.9.

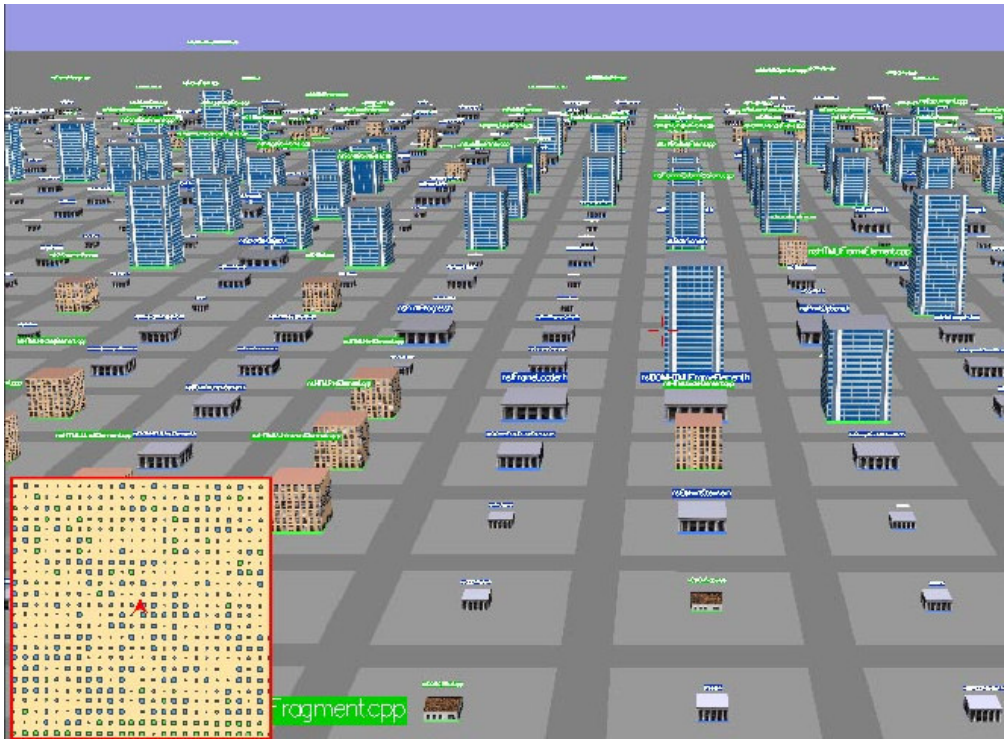


Figure 3.9: The File City - Various Glyphs for Node Representations [13]

3.5 Edges Visualization

One of the basic diagram elements are the links among nodes. Holten [52] came with the alternative representation of edges, which should help reduce the visual clutter and can thus help users to orient easier. They developed five

representations which combine the shape of the edge as well as the colour.

To evaluate the proposed representations, they performed a user study which leads to following recommendations:

- Standard arrow representation (part (a) in Figure 3.10) should be avoided, because the performance of the users is quite low while using it. It is probably caused by the arrowheads, which cause occlusion problems and visual clutter.
- The best results was measured while using the tapered representation (f) in Figure 3.10 for directed graphs.
- For intensity based representation the dark-to-light representation is better than light-to-dark.
- Combining used factors (such as curving, changing colors etc.) for representation of the edges (multi-cue) does not seem to be better than using only one factor (single-cue).

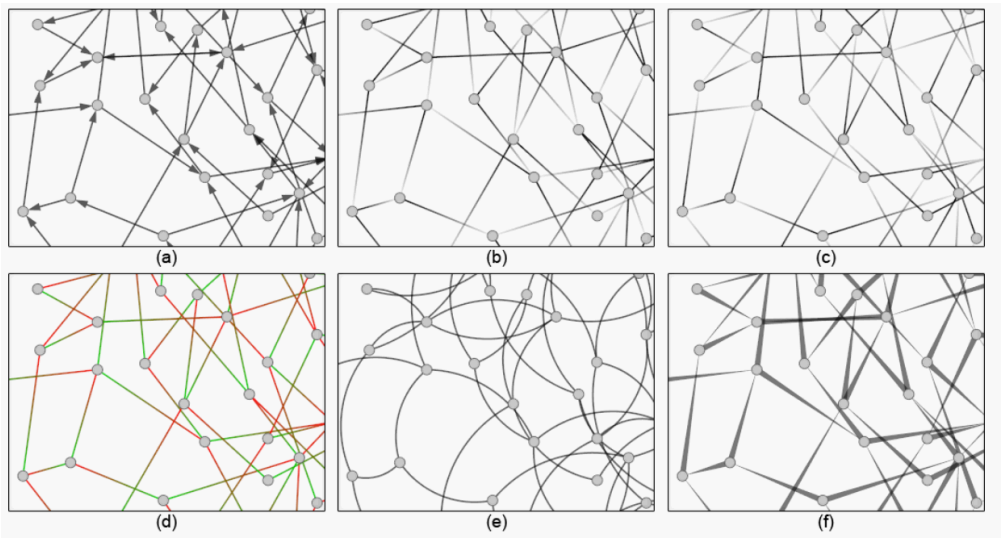


Figure 3.10: The Six Single Cue Directed Edge Representations Used in the First User Experiment. (a) "arrow", (b) "light-to-dark", (c) "dark-to-light", (d) "green-to-red", (e) "curved", (f) "tapered" [52]

For purpose of this work it would be possible to apply above mentioned scientific knowledge on a component diagram. It usually contains directed edges among elements, which represent provided and required interfaces. Also the described intensity of edges can be used for indicating desired component connections metrics. Such as in case of clustered interfaces described in Section 5.5.4.

3.5.1 Edge Bundling

Visualization of large node-link graphs usually suffer from visual clutter. One of the possible solutions of this problem can be using of edge bundling techniques

which can reveal high-level edge patterns. The edge bundling can be applied for both general layouts of graphs and circle layouts. Holten [51] presented self-organizing approach to edge bundling. They model edges as flexible springs attracting each other. They also present rendering techniques to emphasize the bundling.

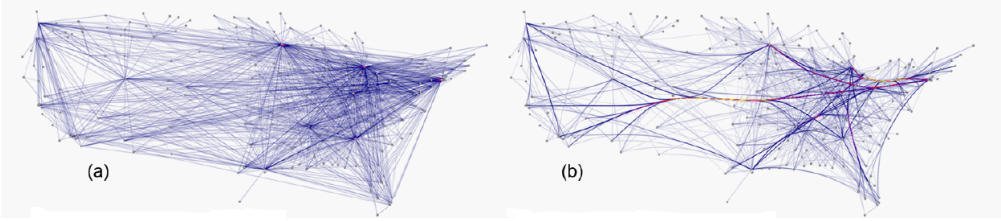


Figure 3.11: US Airlines Graph (235 nodes, 2101 edges) (a) Not Bundled Graph (b) Bundled Graph [51]

Holten [50] also presents a technique of visualizing the elements in circle layout with the possibility to collapse elements. This collapsing and uncollapsing is fully animated, few steps are shown in Figure 3.12. Collapsing leads to replacing of all edges leading from all collapsed elements with one edge.

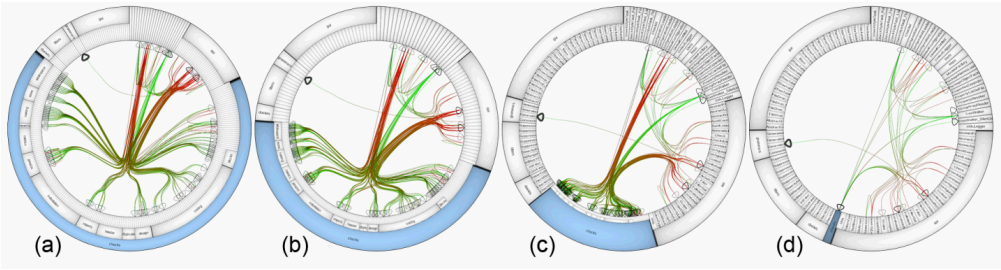


Figure 3.12: Steps of Animation of Collapsing the "checks" Element (highlighted in blue) in (a) Hides All of its Children and Lifts the Relations Pertaining to the Children to the "checks" Element, as Shown in (d). [50]

Also the work of Gansner [40] presented a multilevel agglomerative edge bundling method. It minimizes ink needed to edges representation with respecting constraints on the curvature of the resulting splines. They declare that this method is able to bundle hundreds of thousands of edges in seconds. For comparison they provide the same graph as Holten [51], shown in Figure 3.13.

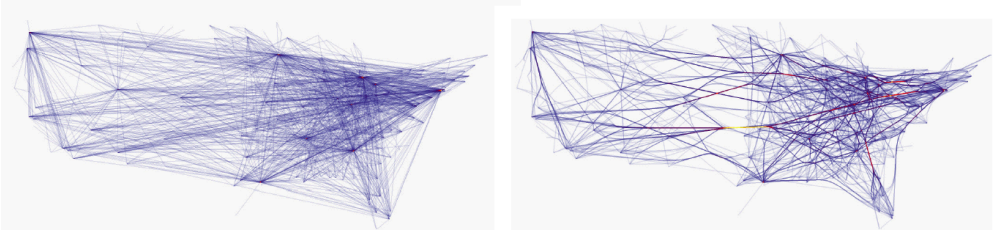


Figure 3.13: US Airlines Graph (a) Not Bundled Graph (b) Bundled Graph [40]

3.6 Background Visualization

The background of visualized node-link diagram can be used for improving the navigation and understanding of visualized system. The work of Byelas [25] presented the tool using the areas of interest (AOI) technique in software diagrams. It investigates correlation of system properties while preserving the layout of displayed nodes. Several rendering modes of this technique are shown in Figure 3.14.

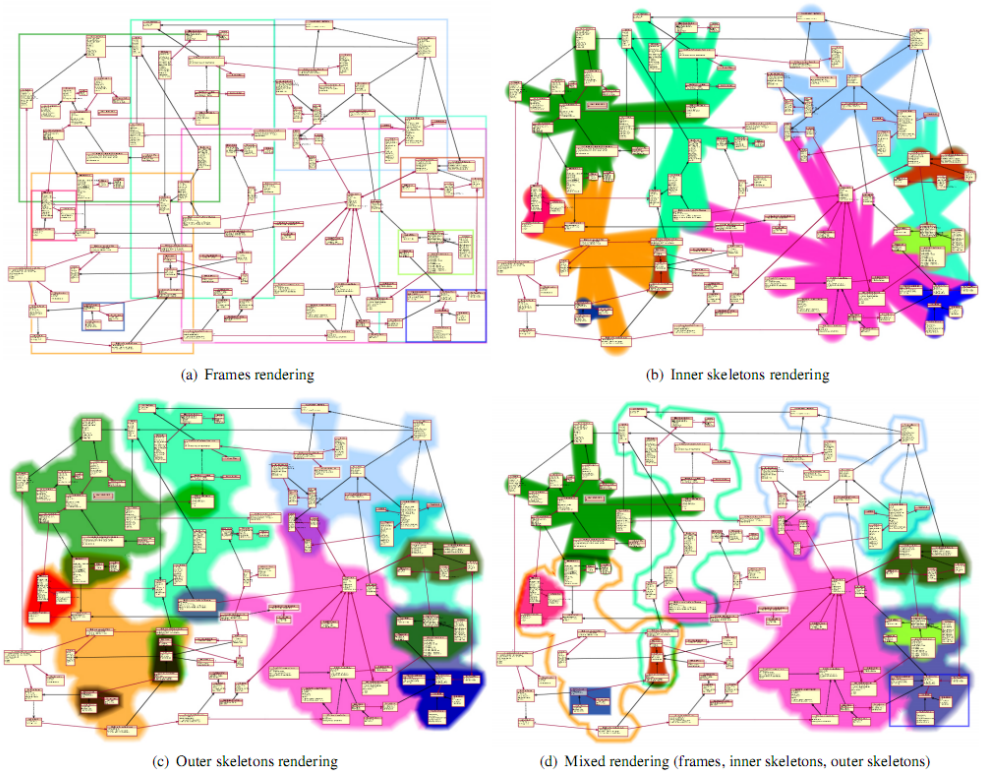


Figure 3.14: UML Diagram with 12 AOIs, Various Rendering Modes. [25]

Another work using diagram background is [39]. It describes the use of geographic maps to highlight clusters and neighbourhoods. Although the work shows the similarities and recommendations arising from TV shows the idea could be adapted for the software visualization.

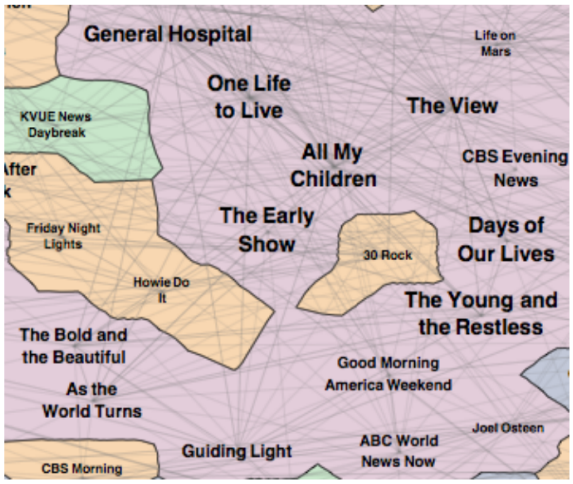


Figure 3.15: Background Maps Used for Displaying Clusters [39]

3.7 Nodes Clustering

Although components usually represent relatively large amount of functionality, we can still find that a particular feature consists of several components. For example a system can have several components for covering the security features of given system. So it is usually possible to find a group of components in a system, which can be considered as a cluster. So the clustering can be used for reducing the amount of components in the displayed diagram.

Cluster can be possibly collapsed into one node, as illustrated in Figure 3.12. Thus the number of nodes in whole diagram would be lowered and the understanding of the whole diagram becomes easier. The number of nodes in the whole diagram is lowered although the connections among components are usually still present and could be shown on demand.

Clusters can either be marked manually, in an automated way [29], [75], [20] or by a combination of those approaches [76]. The overview of clustering algorithms can be found in [93], [111]. While using manual clustering user selects the nodes belonging to a cluster. When using an automated way, a diagram is considered as a graph. In this situation, interconnections (edges) among nodes are usually the most important for clusters creation. To improve the automated clustering we can also use available metrics and information about components. Such information can be e.g., names of the packages (e.g., org.package1.subpackage2). It can help form cluster even though the components are not connected.

Important factor, while choosing a clustering algorithm for certain implementation, is the quality of the clustering result for the given domain. There are several metrics which can help choose the appropriate clustering algorithm stated e.g., in [20]. But even after using these metrics it is usually not clear, which algorithms will give the best result for the general type of graph which component

software application diagram can be.

3.8 Visual Design Guidelines

There are many visual design guidelines, but the basic principle might be summarized as the Visual Information Seeking Mantra [96]:

> Overview first, zoom and filter, then details-on-demand.

There are also described following tasks in information visualization field in [96]:

1. **Overview:** *Gain an overview of the entire collection.*
2. **Zoom:** *Zoom in on items of interest.*
3. **Filter:** *Filter out uninteresting items.*
4. **Details-on-demand:** *Select an item or group and get details when needed.*
5. **Relate:** *View relationships among items.*
6. **History:** *Keep a history of actions to support undo, replay, and progressive refinement.*
7. **Extract:** *Allow extraction of sub-collections and of the query parameters.*

3.9 UML Component Diagram Visual Syntax

There are many software architecture modelling tools and visual syntaxes and their use is very common in practice. As stated in Section 1.3, we mainly focus on UML component diagram [83] in this thesis. We provide examples of other approaches in Sections 4.2.3, 4.2.4.

Most commonly used visual language for displaying component applications structure is UML component diagram. It is generally applicable to most of the component models or frameworks. It captures a basic form of components and their interactions, but advanced concepts like contracts are not included. Visual syntax of its elements is described in Figure 3.16. We can see that it is basically a node-link graph enriched of hierarchical elements and notes as independent nodes. There can be multiple directed edges between any two components (nodes). Any component can be connected any other, which determines, that UML component diagram is not an acyclic graph. As mentioned in Section 1.2, most of the component models are not hierarchical, thus the visual syntax of inner components hierarchy is usually not used. The provided and

required interfaces are basically directed edges, which contain additional information. The main difference, compared to usual directed egde visualization by arrow symbol at end of the edge, is visual syntax showing the direction of an edge by combination of "lollipop" and "socket" symbols at any position of an interconnection line. It is usually shown in the middle of a line, which can make a process of determining the direction of an edge slower in large diagrams. A diagram can also contain provided or required interfaces that are not connected to its respective counterparts.

From the above mentioned we can state that for most of the component models we can consider a UML component diagram as general directed multigraph containing one type of nodes and one type of directed edges. Both nodes and edges contain additional textual information. This gives us the opportunity to apply best practices used in other domains as well as potentially use our new techniques for other domains.

UML component diagram main features are:

- components,
- provided and required interfaces,
- stereotypes,
- tagged values,
- notes,
- hierarchy of inner components,
- including ports as parent's component interfaces.

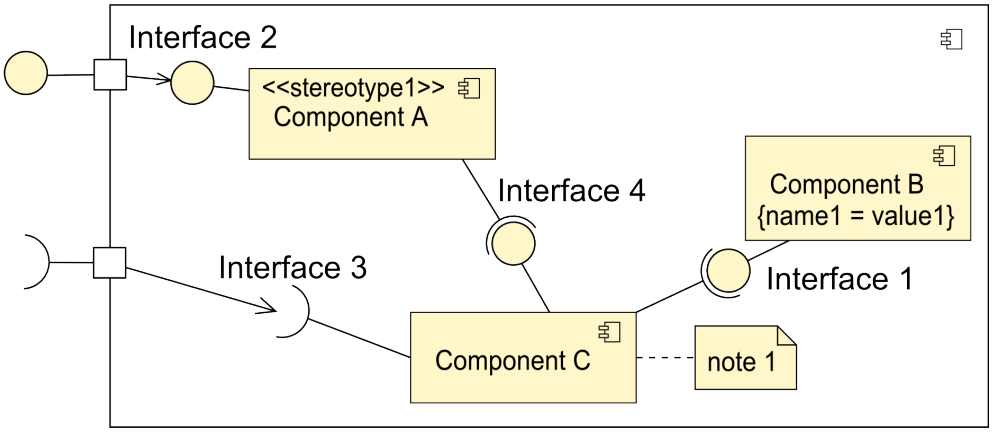


Figure 3.16: UML Component Diagram Example

Chapter 4

Existing Approaches in Component Software Visualization

Most of this chapter was published in [59] and was updated by relevant tools.

4.1 Problems and Approaches to Component Software Visualization

In the following section, we describe the problems in visualization of component-based software related to its diversity, as well as different approaches to visualization of such structures. In Section 4.1.4 we suggest the criteria that can be used for evaluating tools able to visualize such software. These criteria are thoroughly discussed and evaluated from the views of different CBSE stakeholders. The application of these criteria is then presented on the example of IBM Rational Software Architect in Section 4.2.2.

4.1.1 User's Needs and Requirements

People involved in the component development and maintenance process needs to visualize the component applications in various ways. Visualization should help them understand the system, analyse dependencies [71], extract and show desired properties, etc. These techniques are necessary especially when dealing with larger systems which consist from many (hundreds or thousands) components.

Graphical notation is one of the important aspects of visualizing component models. Many component models propose their own graphical notation while other ones assume a generic one like UML; this fragmented landscape can be seen as similar with the situation before UML became widely established for

object-oriented languages. We describe existing approaches including their classification in the next section.

4.1.2 Component Visualization Approaches

Components are by their nature more complex than classes in terms of their contractually specified interface features. Their models, visual syntax, supporting meta-data and tool functionalities should therefore be also more sophisticated. For example, the study [71] shows that architectural modelling would benefit from consolidated views, model consistency and defect checking, and its augmenting by metrics. Additionally, Kollman et al note that obtaining more abstract representations and providing advanced (semantically rich) model features are important for analysts [68].

Several works describe general criteria on analytical visualization tools, e.g. [105] or [69]; both of these works attempt to structure the criteria into categories for better orientation. In [90], there are further identified common desirable features or open issues which can be improved by visualization techniques. Visual notations can be in general analysed or compared from the semiotic point of view, like in [97] or in [81], to understand the suitability of chosen symbols and layouts.

However we are not aware of any other method that would help evaluate component architecture visualization tools. Favre et al discussed several issues with visualization of component-based software in [35]. While Favre covered all areas of component visualization, namely component models, components and their assemblies, he addressed only global issues of such visualization and he did not identify specific visualization tasks, however he provided a solid background and motivation for future work.

The options in modelling and visualizing component software architectures specifically are, cf. [77]:

1. component model-specific tool/notation;
2. generic component-aware tool/notation;
3. UML with profiles;
4. plain UML.

Component model-specific visualization means a visual notation (symbols and their meanings) supported by tools which are able to visualize only one or very few specific models. The motivation for this approach is the diversity of features provided by individual component models. The downside is that the specifics of the given notation can make it difficult for experts from different domains to read and understand the models. Examples of this approach are SaveCCM [44] or Palladio [15] component models.

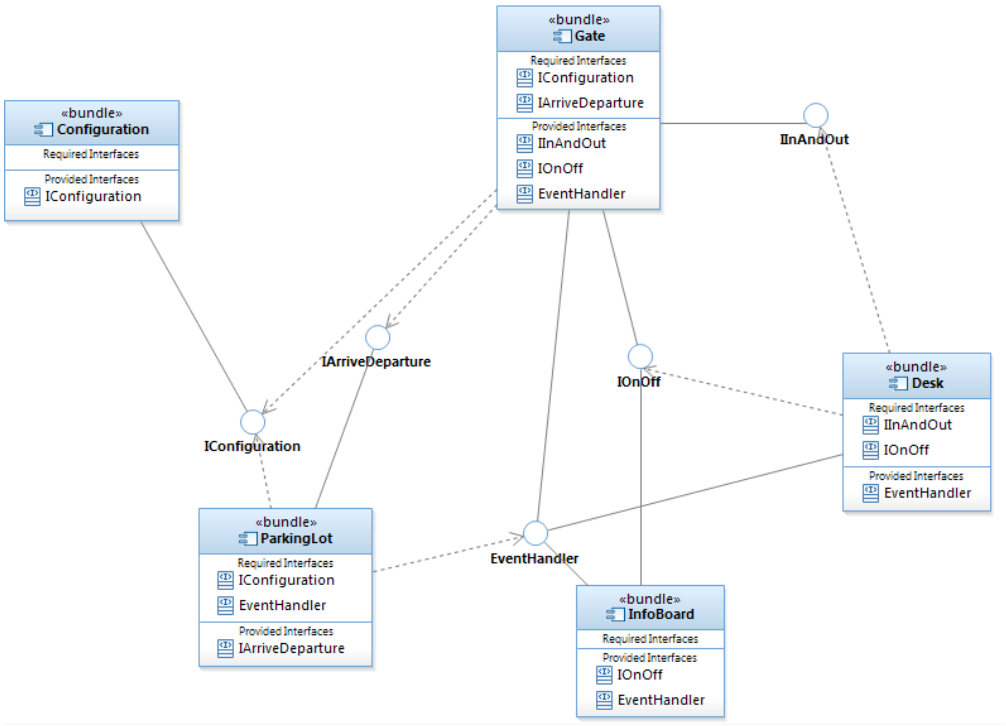


Figure 4.1: Example of Plain UML2 Component Model

Secondly, we can use a universal component-aware visualization tool like SoftVision [104] which is either able to visualize any component model or can be extended for given component model needs. Related to this category is the use of UML [83] constrained by or extended with UML profiles which enable to further specify the semantics of existing model elements and create new ones on top of the core UML meta-model. Creation of profiles including introduction of icons for new model elements is supported by some tools, e.g. IBM Rational Software Architect, and many UML tools are able to use a pre-defined selection of profiles.

Finally, we can use plain UML, especially its component diagram (see Figure 3.16) and possibly class diagram. It may not capture the desired level of details necessary for full component modelling but provides a universal notation that is understood by most software engineers today. Moreover, the tool support is extensive (e.g. MagicDraw, Enterprise Architect, PowerDesigner or StarUML, to name just a few). However, this probably most common modelling approach "...lacks support for capturing and exploiting certain architectural concerns whose importance has been demonstrated through the research and practice of software architectures" [77] and supports only rudimentary analytical tasks.

4.1.3 Problems and Approaches Classification

In general, the options and benefits of a visualization of a component application are affected by: (a) the component model and its features; (b) visual

#	*Functional criteria*	*Category*	*System architect (SA)*	*Compon. developer (CD)*	*Compon. assembler (CA)*
C00	Basic features	N/A	mandatory		
C01	Richness of component interface visualization	Data representation / Static	***	***	***
C02	Model extraction	Integration / Data mining	**	**	
C03	Component and architecture analysis	Data representation / Static		***	
C04	Finding matching variation / extension points	Data representation / Static	***	*	***
C05	Analysis and visualization of extra-functional properties	Data representation / Static	***	**	*
C06	Change analysis	Data representation / Dynamic and Evolution	***	**	*
C07	Analyzing differences between views	Operations / Comparison	**		**
C08	Traceability analysis	Operations / Searching	*	*	**
C09	Model querying and structural analysis	Operations / Searching		**	**
C10	Interactive components clustering	Operations / Searching	***		**
C11	Custom metrics and parameters visualization	Effectiveness / Benefits	**	*	**
C12	Diagram scalability and filtering	Effectiveness / Scalability	**		***

Table 4.1: Criteria and Roles for Component Visualization

notation's repertoire; (c) the capabilities of a tool used for visualization. Suitable visualization approaches have to be general enough to cover a wide range of component models while at the same time being able to capture all aspects of a concrete component model, in order to provide sufficient level of standardization while preserving precious information about the particular component-based applications. In visualization of component-based software it is therefore crucial to provide good notation and diagramming functionalities and beneficial to support more advanced features for architectural analyses, data mining and visualization in general.

In this chapter, we aim to define a suite of criteria that capture these features and emphasize the aspects important from CBSE point of view. These criteria should be suitable for the evaluation of visualization tools to indicate their fitness for advanced visualization of component-based software. Secondly these criteria can guide developers of current or new tools while considering implementation of new features, because each applied criterion increases the added value of the visualization tool.

4.1.4 Criteria for Evaluating Tools

The criteria which we consider important for visualization tools targeted at component-based development are based on the general visualization rules and particular CBSE needs identified in the previous section. The criteria are summarized in Table 4.1; the list is structured using the general scheme proposed by [105] and related to roles specific to CBSE, cf. [103]. Individual criteria are discussed in detail below.

The importance of each criterion for each role is indicated by stars, the scale is from none (not applicable) through one star for lowest importance to three stars for highest importance. Formula 4.1 describes the calculation of final rating s_r of given tool for one role.

$$s_r = \frac{\sum_{i=1}^{n}(w_i \cdot c_i)}{M \cdot \sum_{i=1}^{n}(w_i)} \tag{4.1}$$

Here w_i stands for the criterion importance and c_i represents the coverage of the feature by the given tool, on the scale from zero for "not present" to M for full coverage. Symbol n stands for the number of criteria and M equals three.

Criteria Description

We distinguish between basic and advanced criteria. As basic criteria we consider common tools features, which should be fulfilled in any case. As basic features we consider following:

- pan&zoom,
- diagram overview,

- adjusting the layout of a diagram,
- import&export,
- displaying model structure.

The brief description of advanced criteria follows.

Rich component interface visualization Represents the tool's ability to work with all properties and features specified by component model or framework.

Model extraction Describes the tool's ability to extract model from source code, deployment form or runtime representation, to a representation suitable for working with visualizing the gathered data.

Component and architecture analysis This criterion describes to what degree a tool is able to provide analyses of structures or behaviour of components. There are many possible analyses, for instance for internal dependencies between provided and required interfaces or finding unused required interfaces or structures. Tools can also be able to check architecture style rules, detect design patterns or anti-patterns.

Finding matching variation/extension points A variation or extension point is a vendor defined place for customizing behaviour. The process of finding such point can be very tedious in complex application. But if the tool is aware of the data types and structures it is displaying and is able to run basic queries internally, there is a possibility to offer users a feature which ease this process.

Analysis and visualization of extra-functional properties Extra-functional properties [60] can be either stored in a file or repository separately or can be gathered from the code or running system. Tools can also be able to compose the extra-functional properties of individual components into one property for the system or subsystem, and compare them in order to determine which component is better for a given purpose. There are also several ways of visualizing the gathered data in the diagram or exporting them into another tool.

Change analysis Represents a tool's ability to analyse an impact of the change (e.g. changed interfaces or relations), application consistence and compatibility of given component with other related components after a change.

Analyzing differences between views Although analysing differences in textual data is a common task sufficiently solved by tools, differencing two graphical views is not a very common feature. It enables users to faster understand the changes made in the system.

Traceability analysis Important part of understanding the system is tracing through its dependencies. Although components should be treated as black boxes, composing the dependency along a chain of components from the individual internal dependencies between provided and required interfaces can be very useful. It enables users to predict the ripple effects of potential changes or understand the structure of the system.

Model querying and structural analysis Describes tool's ability to perform user specified or built-in operations which are generally needed to find desired information in the model. It comprises features from basic search to tool's own query language where the queries can be specified by user. Advanced features like structural analysis, model evolution prediction or design patterns and anti-patterns detection are also related to this criterion.

Interactive components clustering Diagrams of large applications become difficult to explore. One of the possible ways of improving the diagrams to be easier to understand is creating clusters of components which semantically represent a subsystem. Clusters can be minimized into symbols to lower the visual clutter of the application's diagram overview. These clusters can be found or suggested by tools automatically and/or adjusted by user manually.

Custom metrics and parameters visualization This criterion describes tool's ability to provide data and related operations, which would lead to visualization of desired metrics and parameters. Important part of this criterion is also the way in which the tool is able to visualize and customize the gathered data. There can be several data sources for the metrics and parameters. They can be stored in a file or repository separated from the diagram representation. Another way of gathering such data can be tool's own metrics measuring and composing capability.

Diagram scalability and filtering In case of large diagrams a tool should be able to handle the load and offer satisfactory response time. This criterion evaluates how the tool handles the problem of model complexity. It can be reduced for instance by multiple levels of displayed details or filtering highly connected parts suitable for detailed view.

In Table 4.1 we can see that most of the criteria are related with the component system architect or assembler and fewer are related with component developers. Component architects and assemblers need to have an overview of the whole system which can consist from hundreds or thousands of components and thus they need lot of analytical techniques and tools to ease their work.

4.2 Tools Implementing Mentioned Approaches

This and following subsections describe the capabilities of current state-of-the-art tools for analysing component applications in view of above mentioned criteria, in the form of a non-exhaustive survey. Primarily it describes the tools which provide interesting features besides basic component diagramming and focuses on those which introduce a novel look on component visualization.

In spite of the imperfections of plain UML component model we briefly present in subsection 4.2.1 selected tools which work with this model as the baseline. We also consider UML profiles as a separate point of view in subsection 4.2.2 because they provide an opportunity to represent various component models. Then we discuss visualization tools specific for some component models in sub-

section 4.2.3 which usually provide very good representation for the given model. Finally, in subsection 4.2.4 we sample tools which are able to represent any component model or at least support a high number of models or languages.

We list the fulfilled criteria from Table 4.1 for each of the tools described in detail. Moreover, we provide overview of criteria fulfillment by tools in Table 4.3.

4.2.1 Plain UML Tools

The UML component diagram describes static application architecture and belongs to the structural diagrams category. It is able to show the components themselves, their provided and required interfaces, associated artefacts and also composition hierarchy by putting (sub-)components inside other components.

There are many tools for drawing plain UML component diagrams, e.g. UMLet[1], Dia[2], VioletUML[3], Gliffy[4], LucidChart[5], Creately[6].

MetricView

MetricView [1] is a standalone tool which allows users to display custom metrics directly in the UML model (C11), as shown in Figure 4.2. Metrics visualization is among others useful for displaying extra-functional properties. This software has a version called MetricViewEvolution which is able to calculate metrics (C05), visualize evolution data (C06) and provide more views for UML model exploration. This tool also implements the area of interest technique [24] for UML diagrams (C12) which helps to highlight areas of concern in the diagram.

[1] http://www.umlet.com/
[2] http://live.gnome.org/Dia
[3] http://alexdp.free.fr/violetumleditor/page.php
[4] http://www.gliffy.com/
[5] https://www.lucidchart.com/
[6] http://creately.com/

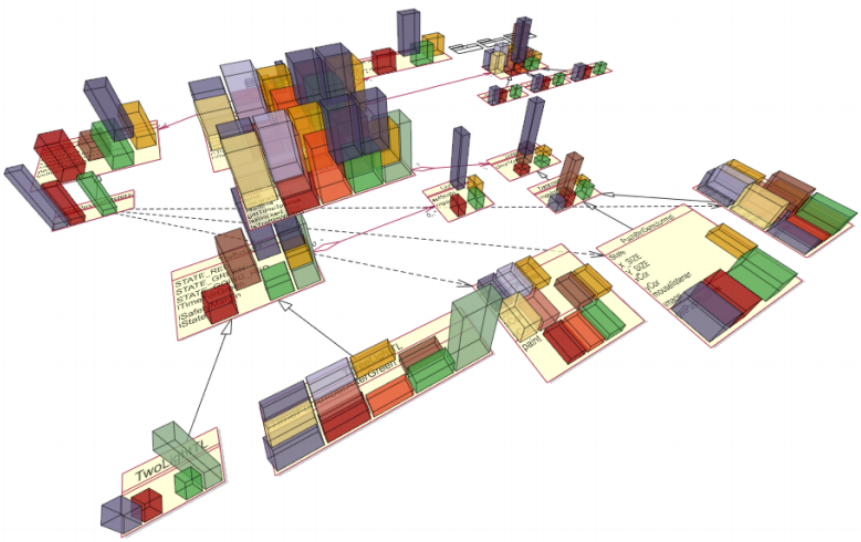


Figure 4.2: MetricView Metrics Visualization

4.2.2 Tools for UML Profiles

For purposes of component application modelling and visualization we can use UML profiles to describe the specifics of component model(s). Visualization and analytical features then depend on the profile support of particular tool.

There are many tools which are able to work with UML profiles such as MagicDraw[7], StarUML[8], Borland Together Designer[9], Visual Paradigm for UML[10] or IBM Rational Software Architect (described in 4.2.2). Diagrams can usually be exchanged among such tools using XML Metadata Interchange (XMI), which should enable UML compliant documents exchange between tools.

Papyrus

Papyrus[11] is the component of the Eclipse Model Development Tools. It is able to work with UML2 exactly according to its definition and supports UML profiles very well. It it is able to customize its editors, model explorer and create user defined perspectives (C11, C12) in a way which provides users the look and feel comparable with domain specific language editors. Papyrus can download the following UML profiles via its update site: MARTE, SysML, EAST-ADL, CCM and LwCCM (C05). The learning curve of this tool can be improved by using tutorials, videos or documentation provided.

[7] http://www.magicdraw.com/
[8] http://staruml.sourceforge.net/
[9] http://www.borland.com/us/products/together/index.aspx
[10] http://www.visual-paradigm.com/product/vpuml/
[11] http://www.eclipse.org/modelling/mdt/papyrus/

IBM Rational Software Architect

IBM Rational Software Architect (RSA) is built on the Eclipse platform. We chose RSA for this case study because it is not just a UML diagramming tool but rather represents a robust solution that supports model driven development, analytical work over different views on the same software and a lot more. All of these features are built on top of the UML meta-model.

RSA offers not only use of UML profiles but it is also possible to design new ones with it. This means that any component model can be represented with details limited only by the UML meta-model itself.

RSA supports all basic features needed for reasonable visualization of component-based software (C00), thus it is possible to use it for these purposes. Richness of contractual levels (C01) is achieved by using UML profiles, extension mechanism which – together with the option to define custom element icons – is powerful enough to model and reasonably well visualize most of kinds of component interface features.

RSA is able to trace dependencies, inheritance or ancestors by using several different features, thus covering the (C08) criteria in its full content. RSA enables model management for parallel development and architectural re-factoring – split, combine, compare and merge models and model fragments, thus (C07) criteria is also fully covered.

For model analysis and model metrics there is a special plug-in, called *The Model Metric Analysis Plug-in* which covers the criteria of (C11). This plug-in enables to create Kiviat diagrams ("spider charts"), perform interactive analysis of model and asses the results. RSA is able to create data sets (queries) to extract a defined set of information from UML models. This feature is accessed by using RSA extended with BIRT project[12], which also enables to create reports and sub-reports. These features cover the criteria (C09).

It may seem that model extraction (C02) is supported, because RSA can reverse-engineer class diagrams from Java, C++ and .NET source code. However, this ability does not work on component-based software and component diagrams. No other criteria are fulfilled.

Evaluation of RSA

Detailed overall value of IBM Rational Software Architect's component visualization capabilities is calculated by using Formula 4.1 and is summarized in Table 4.2.

We can conclude that RSA does not fully cover the desiderata of component application visualization but it still offers features, from which component assemblers can benefit the most.

[12] www.eclipse.org/birt/phoenix/

#	c_i	System Architect	Component Developer	Component Assembler
C01	2	***	***	***
C07	3	**		**
C08	3	*	*	**
C09	2		**	**
C11	2	**	*	**
s_r	12	**0,26**	**0,29**	**0,41**

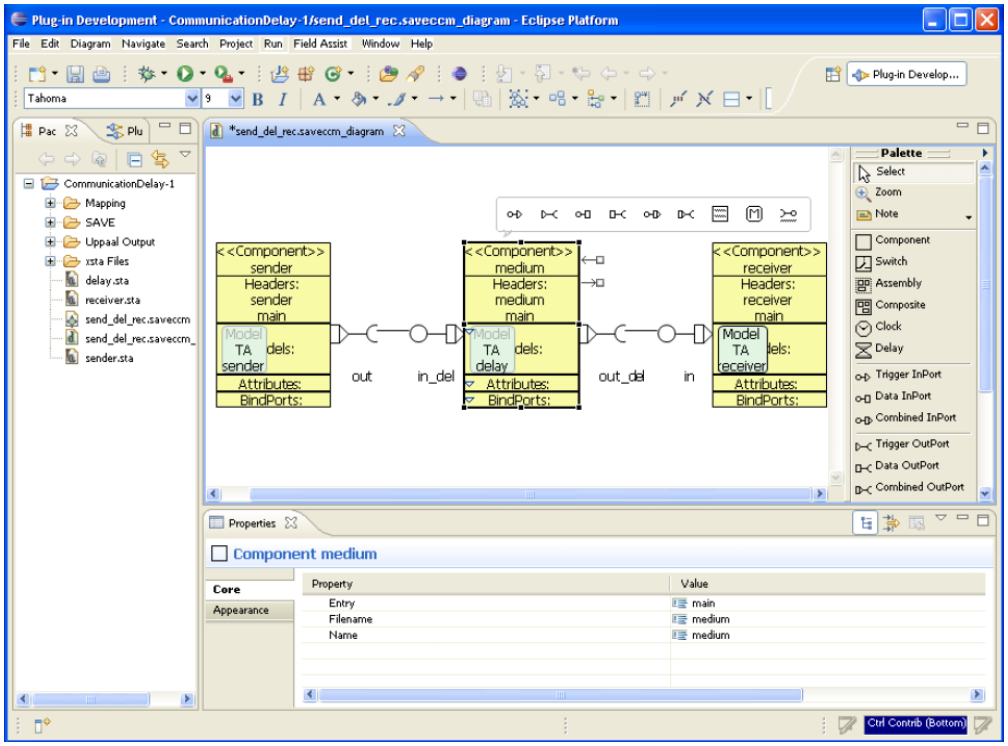
Table 4.2: Assessment of RSA Using Our Criteria

4.2.3 Specific Component Model Visualization Tools

From the tools available for the many existing component models, we selected two representatives with direct support for model visualization.

Save-IDE

Save-IDE[13] is an Integrated Development Environment (IDE) which can be used for the development of component-based embedded systems in the SaveCCM component model. Among others it uses formal specification and analysis of behaviours for designing systems (C05). It also enables internal component analysis (C03). Component visualization in IDE is shown in Figure 4.3.



[13] http://save-ide.sourceforge.net/

Figure 4.3: Save-IDE Visualization

SOftware MOdel eXtractor (SoMoX)

SoMoX[14] is a tool for reverse engineering (C02) of the Palladio component model. Palladio can execute analysis (C05) of software performance, reliability, and maintenance properties on its component-based applications. It is also able to extract the components from source code (C10) written in various languages. Reverse engineering results in the creation of basic and composite components, component interfaces and service signatures, ports (roles), assembly and delegation connectors and behaviour model. The extracted models enable quality analysis and help understand analysed system.

Plug-in Dependency Visualization

Plug-in Dependency Visualization[15] is a plugin for the Eclipse IDE. It enables to extract (C02), visualize and analyse the dependencies (C08) among core and user installed Eclipse plugins, called bundles. The dependency graph helps to understand the system by providing reasonable cognitive support. The user is able to select several options of highlighting the bundles. For example, it is possible to show the shortest dependency path between two selected components. Example of application visualized by this tool can be seen in Figure 5.4.

4.2.4 Generic Component Model-aware Visualization Tools

There are very few tools in this category, and often there is little information available about them.

SoftVision

SoftVision is a software visualization framework described in [104] which is able to interactively explore relations between data structures, as shown in Figure 4.4. It can scale the model to visualize large complex datasets (C12).

This tool enables users to define the structure of the component model used in a given component based system and thus visualize any component model (C02). If the user needs differ for each component model, SoftVision provides elements customization (C11). Thanks to this feature the user is able to create applications which suits well for exploration of given architecture. It also enables to write a custom scenario model which helps users better analyse the system by creating custom map, edit and filter operations (C09).

[14] http://www.palladio-simulator.com/tools/add_ons/somox/
[15] http://www.eclipse.org/pde/incubator/dependency-visualization/

#	*Metric-View*	*Papyrus*	*IBM RSA*	*Save-IDE*	*SoMoX*	*PDV*	*SoftVision*
C01			*				
C02			*		*	*	*
C03				*			
C04							
C05	*	*		*	*		
C06	*						
C07			*				
C08			*			*	
C09			*				*
C10					*		
C11	*	*	*				*
C12	*	*					*

Table 4.3: Tools and Criteria Coverage

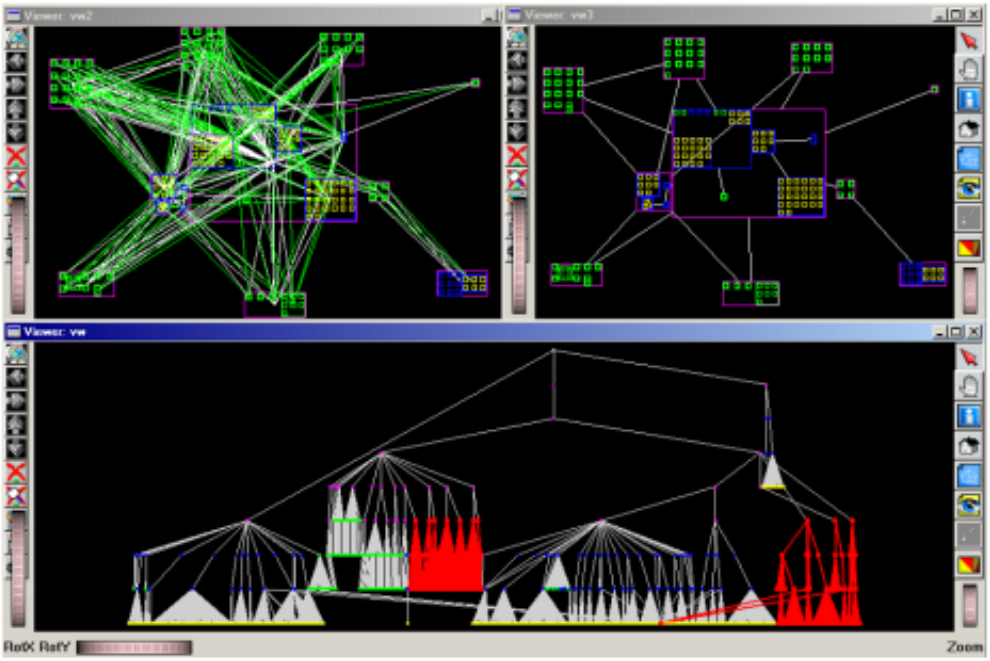


Figure 4.4: Softvision Visualization [104]

4.3 Summary

In this chapter we described four options in component visualization. We provided examples for each of these options. Furthermore, we also suggested several criteria for evaluating tools targeted at component visualization, which can be

applied on existing visualization tools. To summarize the options in modelling and visualizing component software architectures from various points of view, we provide Table 4.4. It describes following criteria:

- how well is a visual syntax known for each approach,
- to which degree is approach supported by tools,
- how much is each of the approaches able to capture all component modelling features,
- how much work is needed before a user can start using such approach.

Sec.	Approach	Visual syntax	Tool support	Compon. features fully captured	Simplicity of preparations before use
4.2.1	Plain UML component diagram	** well-known	*****	*	*****
4.2.2	UML profiles	**** tool dependent	***	****	**
4.2.3	Specific component model	model dependent	model dependent	***** model dependent	*****
4.2.4	Generic	custom	*	** tool dependent	**

Table 4.4: Comparison of Approaches to Component Modelling

Chapter 5

Complex Component Applications Exploration

Techniques presented in this chapter were published in [56] [57] [53].

5.1 Designing a New Visualization

We established the goal of this work in Section 1.3. After that, we described already known techniques and related work in Chapter 3. Furthermore we provided the evaluation of existing approaches in the field in Chapter 4. Above mentioned allowed us to better understand the problems in the field and gave us higher confidence while designing new approach. To ensure the result of our new approach will help reduce mentioned problems and challenges, we used visual design guidelines described in Section 3.8. These are further elaborated in Section 5.3. We also considered using the hardware support for improving the insight into the data shown, as described in Section 5.2.

We identified the current problems in large component visualization (in Section 1.2). Noticeably, standard UML diagrams and their implementations in the industrial tools can depict only diagrams of certain level of complexity. When the complexity rises above this level, the diagrams become no longer visually understandable and start to hinder analytical reasoning. This is mostly problem of diagrams created during automated reverse engineering processes. In this section, we would like to design a new visualization approach, which will cover these problems in a way stated below:

- Diagrams ... become **too big to keep orientation**... - We can simplify the diagram to be small enough to keep orientation without losing the information. We will thus provide a user a way of dividing a diagram into logical parts. Showing these parts (not their content) can provide a high-level overview of given system for a user and help him/her get the initial orientation. Details of the respective parts can be then discovered

interactively. The level of interactivity must not create impediments in exploration. We cover this problem mainly in Section 5.5.5.

- It is difficult to **trace dependencies between distant components**. - When tracing dependencies, usually lines, a user is distracted by other lines or nodes. Moreover, the dependencies are very often between components, which are placed in distant places in a diagram. Thus a user can easily lose the track of given line when scrolling or moving via diagram area. We would like to help a user to show the dependencies in way that he/she does not need to concentrate intensively on such tracing. Thus we provide the concept of showing dependencies without lines, which is described in Sections 5.5.1, 5.5.2, 5.5.3. Additionaly we provide an interactive highlighting techniques (described in Section 6.1).
- ... **how to reduce visual clutter caused by the large number of elements and connections between them**... - Visual clutter is present especially in high-level views. We can reduce the number of visualized elements, but we have to provide a way to show them on demand. We will focus on such techniques, which offer significant clutter reduction and do not require high interaction overhead at the same time. One of these techniques is the interface clustering, mentioned in 5.5.4. Another one helps reducing clutter by removing the most connected components from a diagram area (mentioned in Section 5.5.1).

We can see that above mentioned problems and their potential solutions are highly interconnected, thus we provide an integrated concept of their particular solutions.

The design of a new approach is also driven by two sub-goals and one approach defined in Section 1.3:

- sub-goal: **... not to modify the visual representation of UML component diagram rapidly to shorten the learning curve of potential users.** - Inventing a new visual syntax can help for particular component model, as mentioned in Section 4.2.3, but for shortening the learning curve and allowing wide spreading the approach, we will make only necessary changes to UML component diagram visual syntax.
- sub-goal: **Visualize a sufficient amount of detail to be able to exactly identify individual components.** - This should ensure the practical usability of the whole approach.
- approach: **We will provide ways to hide less important details and show them on demand.** - It should help to lower the visual clutter generally, which is one of the main problems in scalability.

The main approach of this thesis lies in both using known techniques and using novel invented techniques for reducing complexity of large diagrams, which are

generally node link graphs. There are several relatively independent factors when visualizing complex software structures. By improving each of them and combining them together we achieve large increase of the insight into visualized system. The overall picture of the influencing factors important for the scope of this work is shown in Figure 5.1.

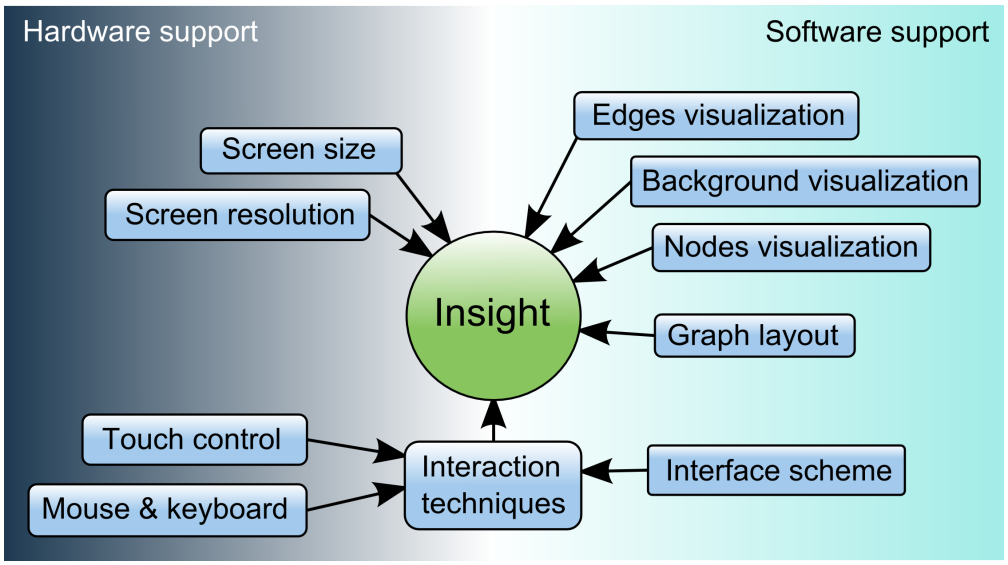


Figure 5.1: Factors Influencing Visualization in Scope of This Work

Visualization techniques which handle complexity, such as off-screen rendering [37], can also help understand a diagram, even it is complex. This chapter describes a novel approach called *Complex Component Applications Exploration* (CoCAEx) which attempts to reconcile the above mentioned contradictory requirements and helps explore the dependencies among components in an intuitive way.

5.2 Using Large Projection Areas

The main idea of improving the diagram understanding by large projection areas is the fact that a person is generally able to see larger area than nowadays standard screen size. Thus the goal of this section is to provide an overview of current projection possibilities with respect to cost of final solution, which would increase the comfort of displayed diagram understanding.

Enlarging the projection area can be easily achieved by using projectors, but we also need to have high resolution to see the details. On the other hand, a very high resolution on small projection area will not bring additional advantages. Thus one of the main requirements for the solution is to preserve reasonable ratio between the pixel size and projection area while covering whole user's perspective.

There are possibly following main ways of achieving the large viewing area:

- using high resolution projectors or monitors,
- using several projectors composition,
- using several monitors composition.

These solutions vary in costs, quality and comfort. While using multiple screens or projectors there is a possibility to use multiple interconnected computers as signal source. In case we have these interconnected computers we can save the costs for the graphic adapters necessary while using one computer. On the other hand, there are further complications while using multiple computers such as delays or data throughput.

In the composition of projectors we can theoretically achieve very large resolution, but currently offered products can provide hundreds of millions pixels. While using projectors, it is possible to achieve projection areas without seeing any visible grid. Current projecting devices for affordable price still do not exceed the abilities of human's eyes in resolution criteria. The eye cannot be simply compared to projection devices by using only resolution metric, because there are many influencing factors in human's reception. On the other hand, we can consider values between hundreds of millions and thousands of millions of pixels as roughly comparable with human's abilities.

5.3 General Design Concepts

To assure our new approach will be designed properly, we followed the visual design guidelines mentioned in Section 3.8. In this section, we describe application of such guidelines on particular concepts of the approach. These concepts are then also covered by the features in our approach in Table 6.1.

1. **Overview** We will show a diagram of reverse-engineered application as an overview first.

   Important part of perception of the application architecture, when using a diagram representation, is the used layout. There are many layout algorithms known (see Section 3.3), they calculate nodes positions based on connections or given metrics. These are suitable for initial load of a diagram, thus an automated layout will be used in the initial step to reduce the visual clutter.

   When creating a mental model of an application a user can have various needs according to which he/she adjusts the layout manually. The manual part of layout creation can take very long time. That is why a good support of custom manual layout is important.

2. **Zoom and Filter** After showing the overview, the diagram will still likely be cluttered. There are many actions in a reverse engineering process that can be automated and thus save the time. Considering that we

have a general component diagram with almost no additional information (just component names, their interface and their interconnections), we can automate only some actions to guarantee that given automation will provide reliable information.

We will provide automated filtering techniques to reduce some part of the clutter out. Zoom features will be available at any time. We will also design our approach in a way that zoom should not be necessary for the first clutter reduction. It will primarily help manual refinements after most of the visual clutter was reduced.

3. **Details on demand and Relate** When having a large diagram one of the key problems (as mentioned in Section 1.2) is showing both details and overview at the same time. If we hide details, we should be able to show them quickly based on the user needs. We have to deal with large amount of interconnections (edges) and components (nodes).

   This guideline can be applied on showing details about the interconnections between two components on demand and thus reduce the clutter caused by number of visualized edges (interconnections). To keep the information about interconnections available, we will provide set of interaction techniques. Even most of the clutter is reduced it can still be large enough to trace the dependencies manually. Thus we will provide better techniques for exploring relationships among components. So the user does not lose the concentration and does not forget the context.

   For the nodes (components) clutter reduction, we would like to apply the details on demand principle as well. We will use groups that can serve as one of the main means for enabling a diagram simplification. They usually identify semantically connected components. A user can use them to create a macro structure of explored system according his/her needs. Having them will allow us to reduce the amount of visualized elements. They are not necessarily equal to functional parts of the system and thus cannot be done in an automated way. Without using them a user would likely use moving individual components to particular places in the diagram area that would have the meaning of the groups.

   A group can in the component application diagram represent:

   - a particular feature consisting of more components,
   - a third-party library used in given system,
   - components developed by particular developer or team,
   - components which are required for another component (or another group),
   - etc.

4. **History** We will provide undo action for defining the set of components that are shown in the diagram. It will be possible to revert the initial visual clutter filtering.

5. **Extract** "Once users have obtained the item or set of items they desire, it would be useful to be able to extract that set and save it ... " [96]. Our approach will allow a user to save the diagram and share it to the other users or mark as publicly available. Saving a custom made layout would allow a user to faster explore the application in case a reverse engineering process takes more sessions (e.g. days).

5.4 Motivation for Clutter Reduction Approach

In a lot of situations, there are components in the system which are connected with large number of other components. If we are able to remove such ones from the diagram area and make them accessible in a different way, it would help lower visual clutter.

Table 5.1 shows several systems (Nuxeo[1], CoCoME[2], OpenWMS[3], Eclipse[4]) with components having large number of connections. The table lists each system per a line with columns denoting the number of components, total number of both clustered and non-clustered connections among the components respectively. While non-clustered connections represent UML-like drawing separately connecting each individual provided-required interface pair, clustered connections collapse all connections between two components into two sets: all provided interfaces and all required interfaces.

Several experiments using the proposed technique were performed, based on the data in the table. In one of them only 7 Nuxeo components have been removed from the diagram area leading to 241 and 431 lines remaining in the graph for the clustered and non-clustered versions, respectively. Therefore, the graphs were reduced of of 69% of lines in the clustered and 65% of lines in non-clustered version.

These numbers show that using the proposed technique, can achieve significant visual clutter reduction. Visual effect of the results is shown in Figures 5.2 and 5.3, using circle layout for clarity.

System	Components	Clustered	Non Clustered	Clustering Effect
Nuxeo	202	698	1425	48%
CoCoME	37	125	188	66%
OpenWMS	65	232	642	36%
Eclipse	378	533	1079	49%

Table 5.1: Several Systems with the Number of Components and Connections

[1] http://www.nuxeo.com/
[2] http://www.cocome.org/
[3] http://www.openwms.org/
[4] http://www.eclipse.org/

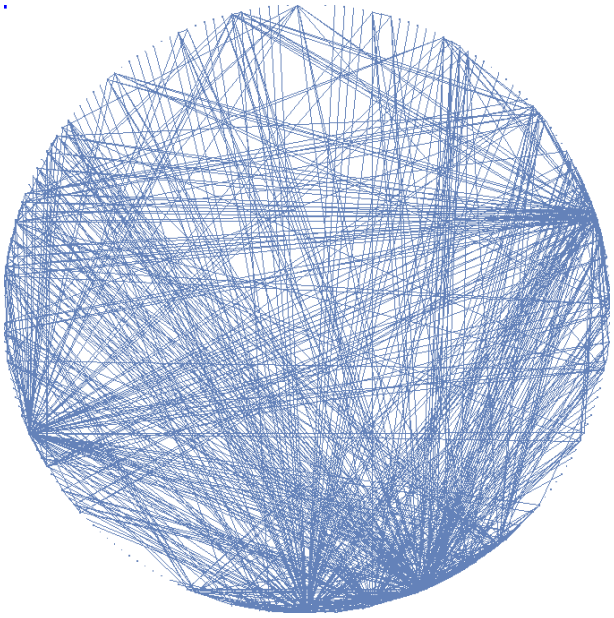


Figure 5.2: Nuxeo Before the Reduction

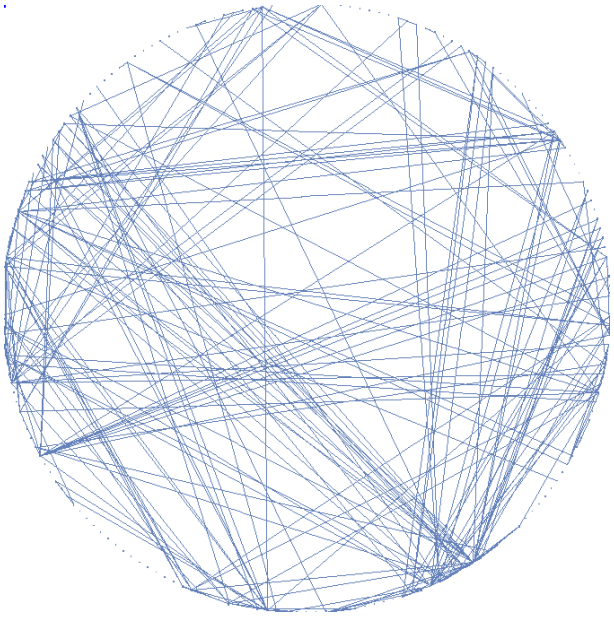


Figure 5.3: Nuxeo After the Reduction

5.5 Techniques for Lowering Visual Clutter

In this section, we describe the problem of the visual clutter first. It also focuses on the problem with highly connected components and the clutter caused by their connection visualization. These components highly contribute to the visual clutter of the displayed area. As an answer to one of the main problems ("how to reduce visual clutter") described in Section 5.1, we present a novel

technique that helps reduce the visual clutter in large graphs.

Very often, only a small amount of components is connected to a large number of other components. It results in a lot of lines going only from few components as can be seen in Figure 5.4, where is shown part of Eclipse[5] structure in Plugin Dependency Visualization tool[6]. Such components are often, among developers, informally called "God Objects". Having such objects (components), the user is limited in recognition of other connections in their surrounding area and trace the connections themselves. Another side effect of these components is that they fill a lot of space, thus exhausting one of the essential resources in the visualization which can be used for easing the work with large component diagrams.

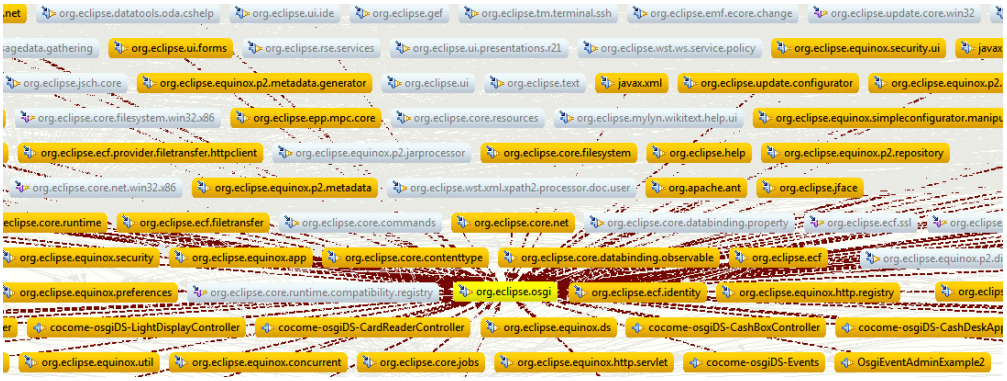


Figure 5.4: Wide Amount of Lines From One Component

Another very common situation is that a user is not able to trace the dependencies among components due to large amount of lines in the whole diagram. Figure 5.5 shows UML diagram of CoCoME[7] application, which has 37 nodes and 244 connections. We can see that the diagram is cluttered and tracing dependencies would be difficult.

[5]Popular IDE, see http://www.eclipse.org/
[6]http://www.eclipse.org/pde/incubator/dependency-visualization/
[7]http://www.cocome.org/

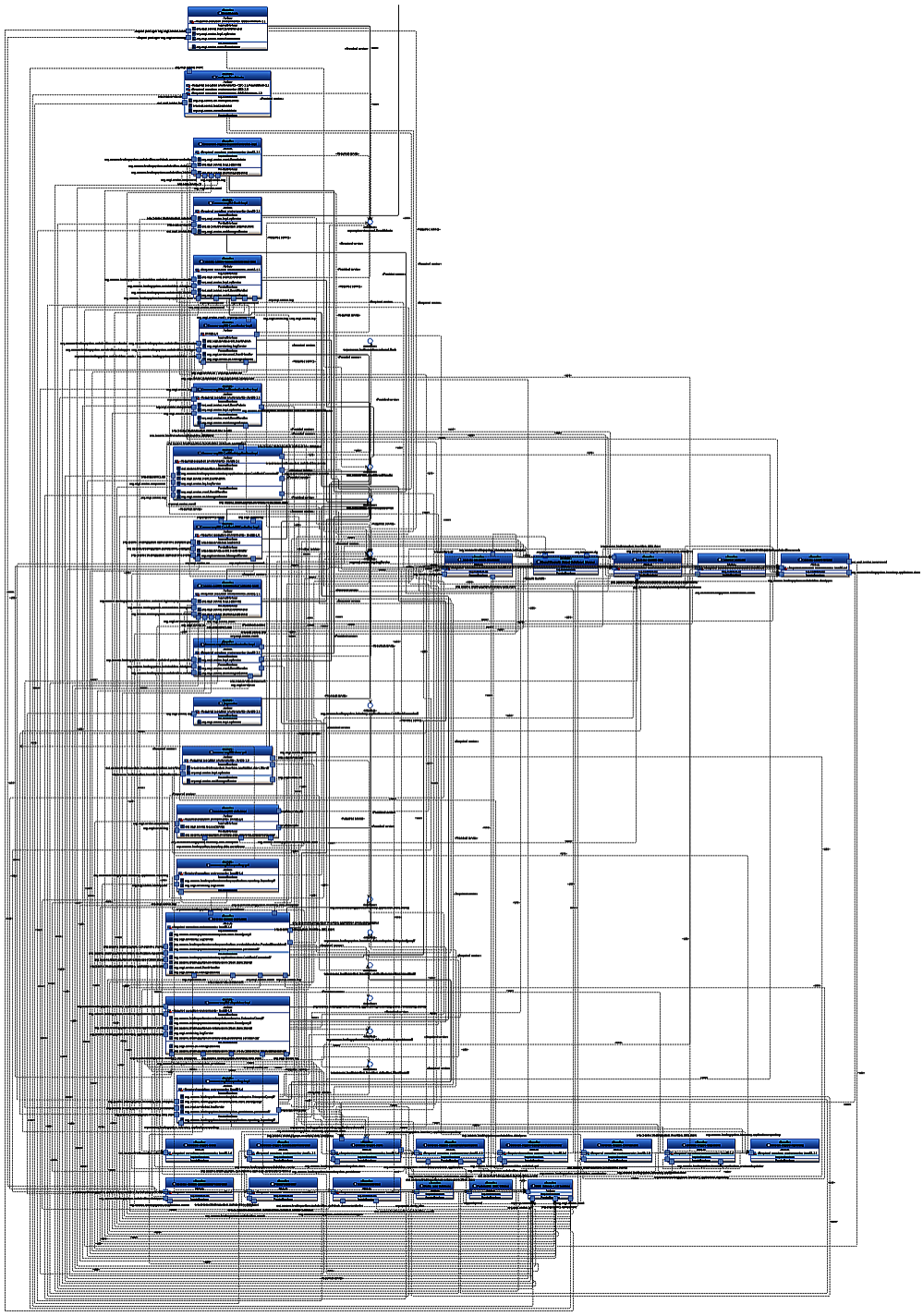


Figure 5.5: CoCoME Application Visualized with UML [98]

The key design concept which we apply to reduce visual clutter is to remove highly connected diagram nodes from the diagram itself and place them in a separate view linked with the main diagram. This achieves cleaner diagram by removing the related edges (which cause the clutter). We call the view *separated components area* (abbreviated to SeCo). It is placed on the border of a window.

Placing the component in the SeCo essentially marks the component as a "familiar one". The user may then concentrate on and continue getting familiar with the rest of the system. It is an extra area besides the standard diagram area that displaces the components with the high number of connections (and thus lines).

After moving components from the main diagram to this area, the lines between these components and remaining components are elided. Instead of them a representing visual symbol is used in the diagram area. The same symbol is shown near the removed component in the SeCo. It reduces the number of lines in the graph not reducing the information provided. Obviously, components with a high number of connections are the most beneficial to be moved, because they reduce the high number of lines from the graph. For instance, a user may displace a component implementing a logger. Such a component is probably used by most of components in the system and its displacement reduces the graph complexity.

We assume both automatic and manual component selection may be used. In the automatic case, all components with the number of connections overcoming a certain threshold are displaced. In the manual use, a user selects the components to be moved to SeCo directly in the diagram area.

In the following sections, we describe in detail the individual parts of the whole visual design used by this technique.

5.5.1 Separated Components Area (SeCo)

SeCo is a part of the application window. It can be placed on left or right side of the window, because current screens have wide aspect ratio and thus using these sides will not deform the rest of the viewing area as much as using the top or bottom side. The wireframe of the application window is shown in Figure 5.6.

Generally, there is also a possibility to show SeCo as a floating element in the diagram area. A drawback of such solution is that a user can easily forget the place in a diagram, where the SeCo is located. It would also be an element, which affects the layout. That is why we chose the fixed version, which we also find more intuitive and easy-to-use.

The content of the SeCo is described in following subsections.

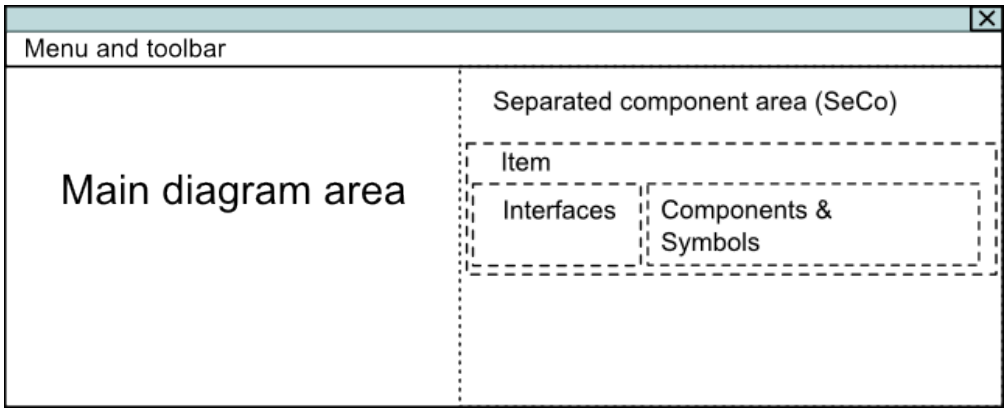


Figure 5.6: Overall Layout of the Application Window

5.5.2 Items

SeCo consists of a list of items and *unconnected components list*. Each item consists of components, interfaces and one corresponding symbol (see Section 5.5.3). Components placed in SeCo have displayed relations with the rest of the components in the diagram on the border between diagram area and SeCo. The interfaces are always shown near the border between diagram area and SeCo.

We distinguish between two situations corresponding to an item's internal layout of components and the representing symbol. In the first situation, when there is only one component in the item, interfaces are directly connected to the component and the symbol is behind the component as shown in Figure 5.9. In the second situation, the item consists of more components which form a group. In this case, the interfaces are directly connected to the symbol and the components are shown behind the symbol (Figure 5.12). The former situation stresses the display of the interfaces-component connections while the latter situation stresses the space saving. Groups are described more in detail in Section 5.5.5.

5.5.3 Symbols and Delegates

The purpose of symbols is to create clear and easily recognizable key which uniquely identifies one item within SeCo. Symbol should be small enough to save space anywhere it is used. The user should be able to choose its own symbols. We have chosen letters for the demonstration of the idea, but it can be any other symbol or an icon. Examples of symbols are shown in Figure 5.7.

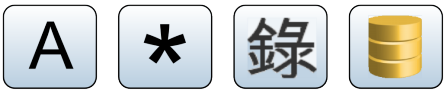


Figure 5.7: Example Symbols

To solve the difficulty of tracing dependencies (as one of the problems defined

in Section 5.1), we propose following concept. To keep the information about the connections in the diagram area when lines are removed, we use so called delegates. They represent the connection between given component and the corresponding item placed in SeCo. In the diagram, they are shown as small rectangles neighbouring the displayed components and containing the symbol, which corresponds to the connected item (see Figure 5.8).

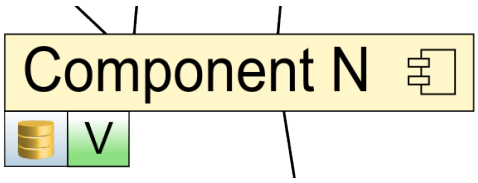


Figure 5.8: Delegates in the Diagram Area

Showing particular delegates in the diagram area can be toggled by clicking on the symbols in SeCo. The SeCo item indicates the state when delegates are shown by different colour of symbol's background as shown in Figure 5.9. The indication of the state when delegates are shown could be also changed to show a checkbox, or other graphical element. We chose different background colour in order to save screen space. So the purpose of the background colour is that it clearly indicates whether item's delegates are shown.

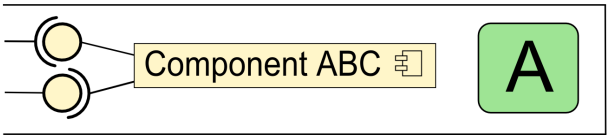


Figure 5.9: Item Design When Showing Its Delegates

Using symbols provides an alternative to using a line between two connected elements. Commonly used approaches of showing such relations are:

- adjacency matrix,
- node-link graph visualization.

5.5.4 Interface Clustering

For each component shown in SeCo, interfaces are clustered into two sets: all provided interfaces (displayed as "lollipops") and all required interfaces (displayed as "sockets"). This is shown in Figure 5.9. It helps to minimize the space which these elements fill. Numbers in lollipops represent the value of a metric associated with the interface group, e.g. simple count, interface complexity, extra-functional properties. For better design illustration, we chose number of interfaces as the metric, which is easy to understand and imagine the concept deeper.

The clustered interfaces are by default not connected to the rest of the diagram by any lines which reduces the total amount of lines in the diagram area. The connections (resp. lines) can possibly appear only when interacting with any

of the components involved or the clustered interface itself.

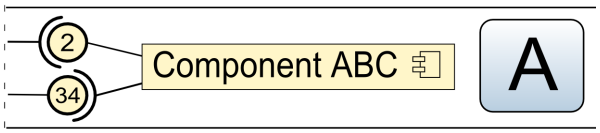


Figure 5.10: Clustered Interfaces

There are two kinds of interaction with clustered interfaces. First is a simple showing the connections lines and highlighting the components involved after user hovers with mouse cursor on the clustered interface. Second is a showing of the details of all interfaces including names, connections and highlighting the involved components. It is launched by mouse click on the clustered interface. It is shown in Figure 5.11 for Interface 4. In a case a component from the diagram area connected to an inspected interface is not visible in the current diagram area view, it does not make sense to show the connection line. Thus a proxy component is shown instead. This situation is shown in Figure 5.11 by the rectangle with rounded corners – Component N.

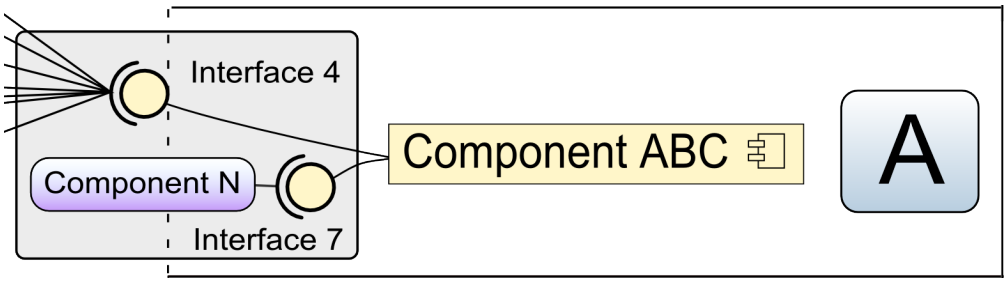


Figure 5.11: Interface Details

5.5.5 Component Groups

Very often a particular functionality is implemented by several components, for example the CashDesk subsystem of the CoCoME which is composed of 4 components. In a case this functionality is used by a large number of other components in the system, it is beneficial to represent them as a group in SeCo.

All components (one or more) from such a group can then be replaced by one delegate in the diagram. It saves space in the diagram and also helps to create semantic groups of components. It consequently improves understanding of the whole system where user may e.g. find cliques of components first. These may be then grouped and displaced from the graph to continue exploration of the remaining graph. Thus we can reduce the number of elements in the diagram area and make a whole diagram smaller which can help solve one of the main problems defined in Section 5.1.

The group symbols visually differ in component symbols and colours. A group symbol is larger in the size compared to the case of a single component, to denote the fact the group shows a large number of components. It shows

two additional categories of clustered interfaces. These categories contain all provided interfaces not used by any other component in a diagram and analogically all required interfaces which no other component provides. In the case of single components it is better to show these interfaces only on demand and thus save the space. The group is shown in Figure 5.12. Showing not used interfaces can easily inform the user about potentially missing components and thus prevent the future deployment problems.

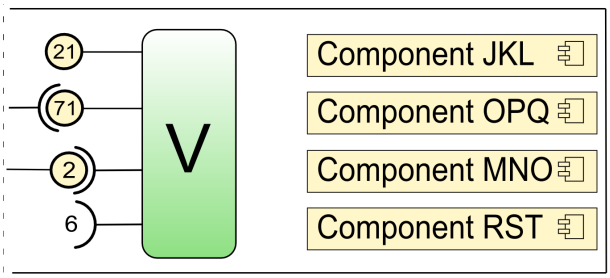


Figure 5.12: Group of Components Represented by a Group Symbol

When showing delegates in the diagram area for a given group or an item, its symbol appearance changes, see Figure 5.13 (group "V"). We have chosen different the background colour for demonstrating this item's state as shown in Figure 5.13 on group with *symbol* "V". This situation is equivalent to the situation of one component.

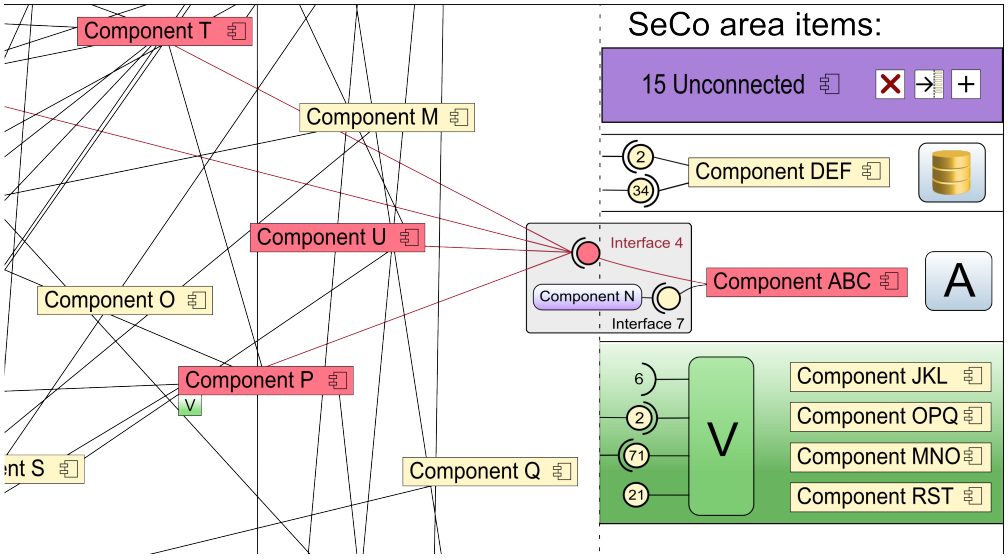


Figure 5.13: Application Layout with an Example Diagram

5.5.6 Unconnected Components

For lowering the amount of components displayed in the diagram area, we created the `Unconnected components` item in the SeCo (see Figure 5.13). There are all components which are not connected to the rest of a system. These com-

ponents are thus not exhausting space in the diagram area. This SeCo item is collapsed, so the content (list of unconnected components) is not visible, as shown in Figure 5.14. It can be expanded on demand by clicking the plus icon. In case of need a user can also move these components to the diagram area either individually by using a red cross icon next to the name of the selected component, or all of them by using the same icon next to the `Unconnected components` label.

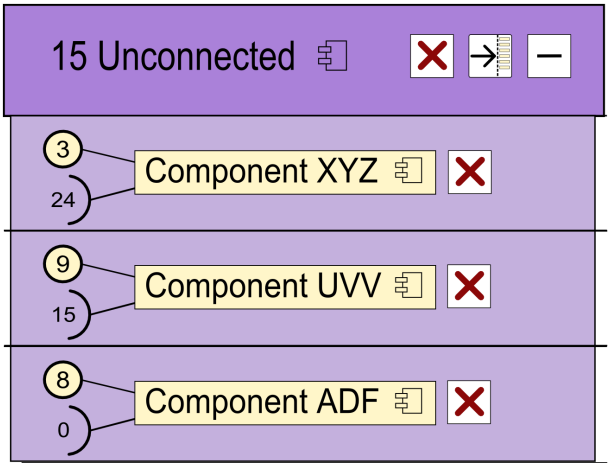


Figure 5.14: Unconnected Components Item Expanded

5.6 Viewport for Component Diagrams

We described the concept of groups in Section 5.5.5, which works with groups mainly in SeCo. As a group can consist of many components, so there are likely many connections with the rest of the diagram. While having the concept of groups we are able to highlight the components connected with a particular group (resp. any of its components). To be able to view internal dependencies inside a particular group we introduce the viewport concept. It should help us to work with the group exploration. The viewport technique should also enable to explore and understand the dependencies in large diagrams by showing the context of a selected diagram subset, because tracing the dependencies leading outside of the screen can be difficult.

The proposed technique called `viewport` shows the graph (standard UML component diagram) zoomed-out to provide the appropriate overview of the complete architecture, with elements displayed without details. Besides that it shows selected components in detail inside a *viewport area* plus all their relations with other components in the diagram in an interactive border area (see Figure 5.15). These relations are for each component clustered into two sets: all provided interfaces (displayed as "lollipops") and all required interfaces (displayed as "sockets").

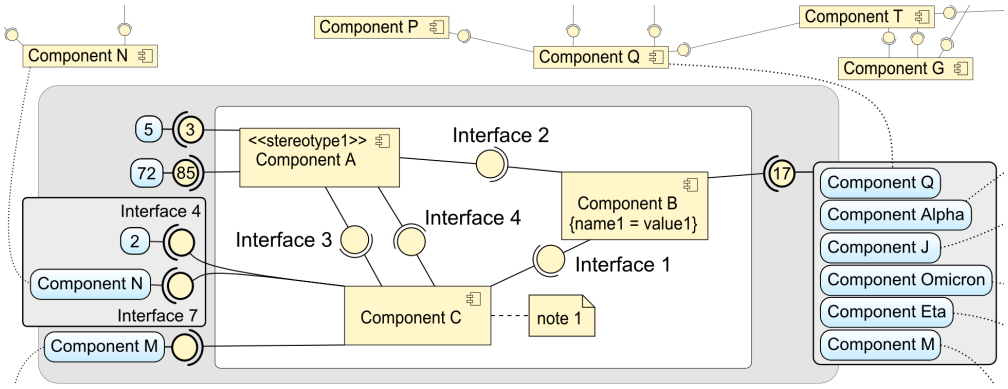


Figure 5.15: Viewport for Component Diagrams

These interfaces are then connected to clustered proxy components, visually represented as rectangles with rounded corners. Each rectangle represents one or more components. Numbers inside the clustered interfaces and proxy components represent a desired metric, e.g. the number of elements clustered in a given symbol. One of the key factors of this approach is the interactivity of the border area. It is important for user manipulation with clustering of interfaces or components, manual or automatic layout adjustments and selecting the components to be shown in the viewport.

The interface clustering shall reduce the visual clutter otherwise caused by large number of relations. The proxy elements should reduce the need for the disorienting pan&zoom otherwise necessary while exploring dependencies and provide user relevant information in one place. The viewport can either be placed on a given position in the diagram (there can be more viewports in a diagram) or have a fixed position on the screen. For our specific use we prefer to use first mentioned, because we find it more powerful and flexible.

5.7 Using the Viewport Technique for Groups of Components

This section presents the usage of the viewport technique for groups (as described in Section 5.5.5) and its integration with SeCo technique. A group of components shown in SeCo can be moved to the diagram area and shown with the viewport technique. Similarly the viewport form of a group and its content can be moved from the diagram area to SeCo.

It is possible to show the group in a diagram in following levels of details:

1. with a viewport technique with all details for all components and their relations in given group,

2. as a symbol belonging to a group only.

These possibilities are described in following sections.

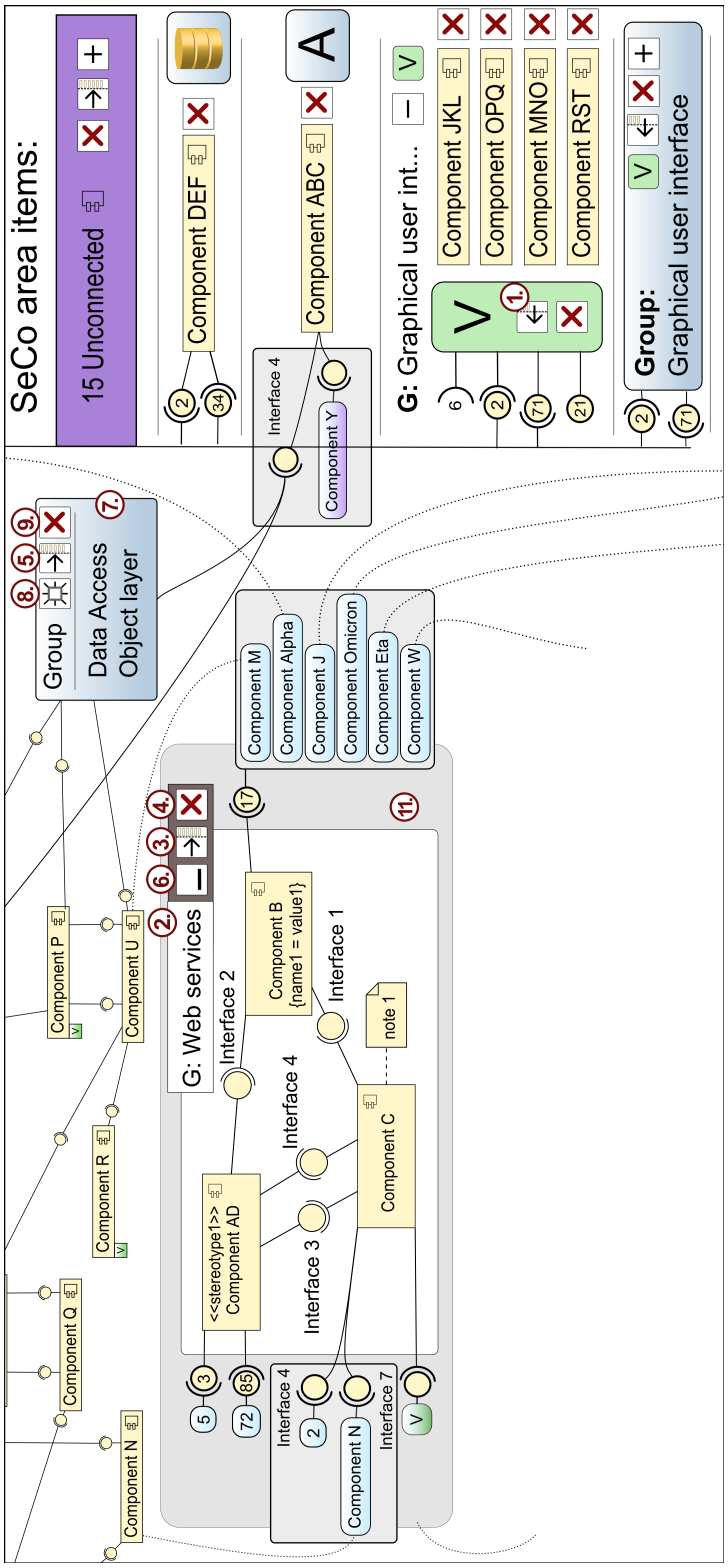


Figure 5.16: Viewport with SeCo

5.7.1 A Group as Viewport with Details

It is possible to move a group from SeCo to the diagram area (by icon indicated with mark (1.) in Figure 5.16). The group will then disappear from the SeCo and will be shown only in the diagram area as a viewport.

Each viewport has its own small toolbar, which contains a symbol representing appropriate group (2.) and icons for important actions. The symbol has similar meaning as symbols used in SeCo. It is possible to cancel the viewport (4.), release its components to the diagram area. Viewport itself is then deleted. Another possibility is to move a viewport to SeCo (3.). It removes the viewport from the diagram area and shows its contents in the SeCo as a group. Finally, a user is able to minimize a viewport (6.). It is then represented as a symbol only, which is described in following section.

5.7.2 Group as a Symbol

It is possible to show a group as a symbol only (marked with (7.) in Figure 5.16). It is very important part of visible elements reduction process as well as visual clutter reduction. Group symbol represents the whole group and its content. It means that components included in the group are not visible at the time the group is collapsed into the symbol.

When a user hovers a mouse over this symbol a small toolbar appears. It contains icons for following actions:

- showing viewport in full details (8.), which shows viewport in a way described in Section 5.7.1,
- moving group from diagram area into SeCo (5.),
- releasing components from given viewport (to the diagram area) and removing the group itself (9.).

5.8 Extra-functional Properties Visualization

As we introduced EFP in Section 2.4, we will now describe its visualization integration into our visualization approach. To fullfil our sub-goal "not to modify the visual representation of UML component diagram", we designed the EFP visualization in a way that uses existing UML elements.

To improve existing visual representation and speed up the EFP problems resolution for user, we decided to visualize EFP directly in the diagram. The above mentioned design was modified in a way described below to enable such visualization. From the point of view our approach, EFP is a set of attributes for each interface. These are then visualized separately on demand and compatibility of particular pair of components is visually enhanced for EFP awareness.

The particular EFP can be shown by clicking on the clustered interface icon. A tree containing EFP definitions is then opened. This tree contains available EFP definitions including their values on both sides (provided and required). The icons at the beginning of a line marks whether a particular property of given connection is satisfied (green circle) or not (red circle), as shown in Figure 5.17.

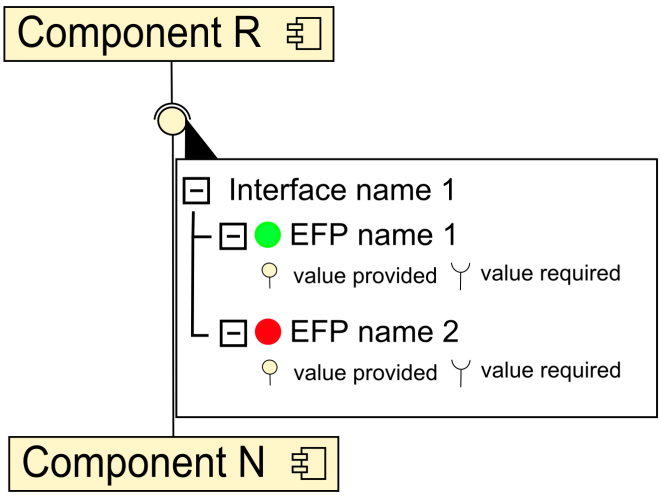


Figure 5.17: Exploration of Clustered Interfaces Enriched by EFP

To enable faster recognition of possible problems in EFP composition, we provide an EFP compatibility indication inside of clustered interfaces icons. The compatible component connections are marked by "green tick" icon. Incompatible connections are marked by "red cross" icon, as shown in Figure 5.19.

Each connection between components consists of required part icon and provided part icon. In case two components are compatible a user can be informed about a degree of their compatibility by interfaces scaling. Both provided and required part of the interface icon symbol are scaled separately according to their degree of compatibility between existing EFP values, as shown in Figure 5.18.

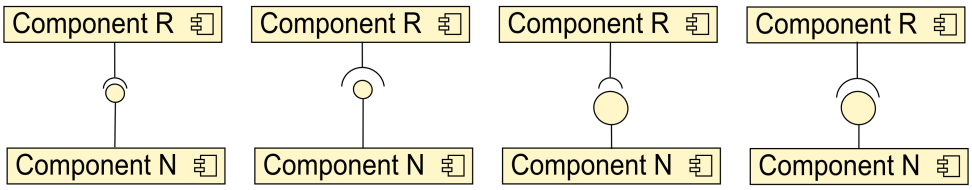


Figure 5.18: Interfaces Scaling According to Relative EFP Values

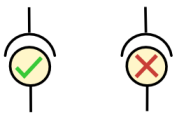


Figure 5.19: EFP Compatibility Visualization

Chapter 6

The CoCAEx Tool: Experimental Implementation of the Approach

Techniques' implementation mentioned in this chapter were published in [54], [53], [55], [63].

This chapter describes implementation of techniques proposed above. There are also some differences in the implementation compared to the design. These are described more in detail in respective sections below. The resulting implementation availability is described in Appendix A.

6.1 Techniques Implementation and Demonstration

We describe resulting implementation of techniques proposed in Chapter 5 in the similar structure to provide the demonstration of CoCAEx tool. Additionally, we provide description of global features. Subsections 5.5.1 - 5.5.4 describing SeCo concept design are covered by implementation in Section 6.1.2. Sections 5.5.5, 5.6 and 5.7 describing the concept of groups are covered by implementation in Section 6.1.4. The main window wireframe of CoCAEx application is described in Figure 5.6, more detailed implementation description and demonstration follows.

6.1.1 Global Features Implementation

CoCAEx global features are described in following paragraph that mentions the marks in Figure 6.1. It provides standard features such as panning and zooming. There are two modes of manipulating the components with appropriate icons in the toolbar. First mode is for moving components (A) where the user can manually adjust the layout of a diagram. Second mode (B) serves for removing components from the diagram area to SeCo simply by clicking on the desired

components that should be removed. Last two icons in the toolbar serve for the automatic removal of a configured amount of components from a diagram to SeCo. The tool is currently configured to remove 15% of most connected components. The icon (C) is used for removing these components and adding them to SeCo as individual items. The next icon (D) creates one group for all of them.

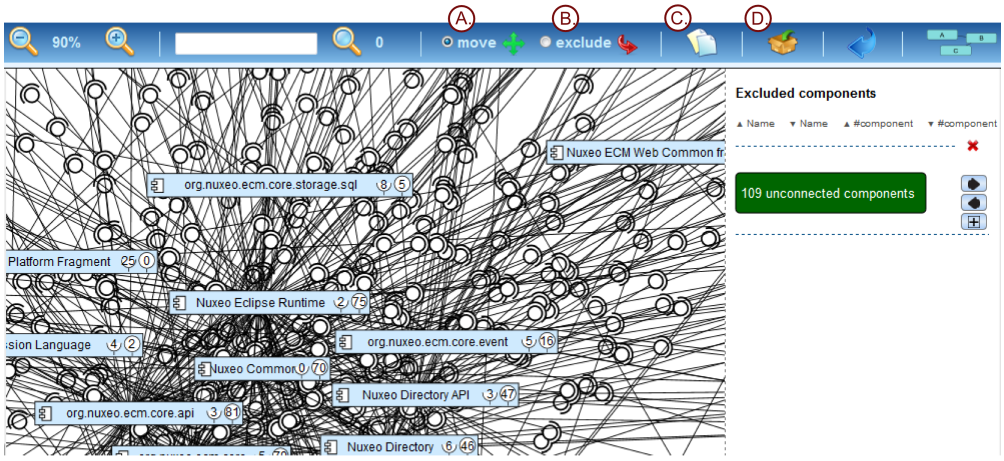


Figure 6.1: Initial Load of Nuxeo System Loaded into CoCAEx Application

CoCAEx offers a fulltext search in components' names. In Figure 6.2, one can see the search for a part of name "org.nuxeo.ecm". Thirteen components contain this name as indicated by the number thirteen next to the magnifying glass icon. Matching components are highlighted by orange colour in the diagram area. This feature is able to highlight components both in the diagram area and in SeCo.

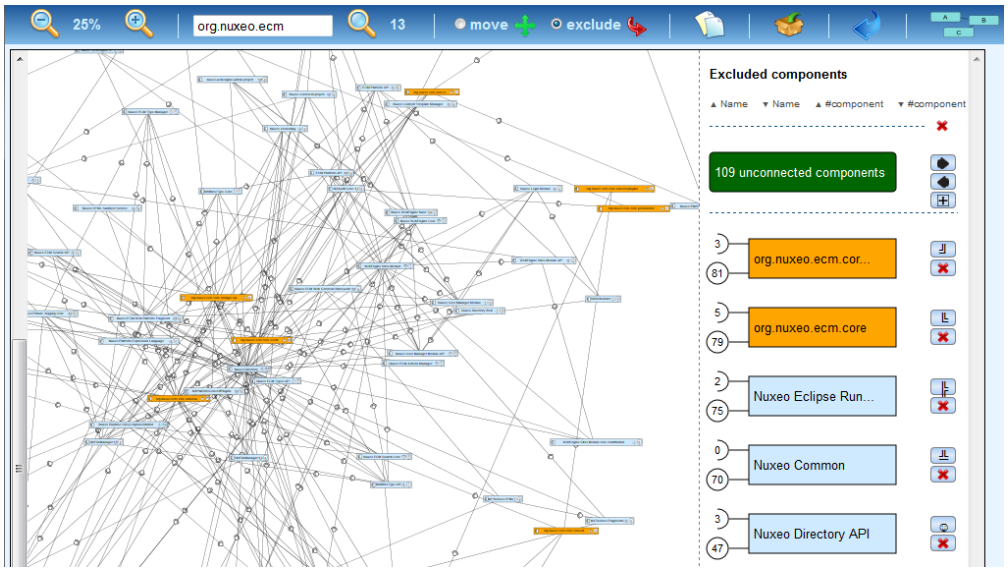


Figure 6.2: Forming Clusters via Search Feature

6.1.2 SeCo Features Implementation

If one clicks on the provided interfaces of a component in SeCo, these interfaces and connected components become highlighted by blue colour. An example in Figure 6.3 shows the dependency between the *Nuxeo Eclipse Runtime* component's provided interfaces and the components highlighted by blue colour. These highlighted components can be either in the diagram area or in SeCo. Similarly, for interfaces required by components in SeCo, highlighting by red colour is used. There is also a possibility to click on the component itself and highlight both required and provided interfaces. In a case when a particular component is connected to the explored one both by provided and required interfaces, red colour is used for highlighting. This can be improved in the future to use different colour.

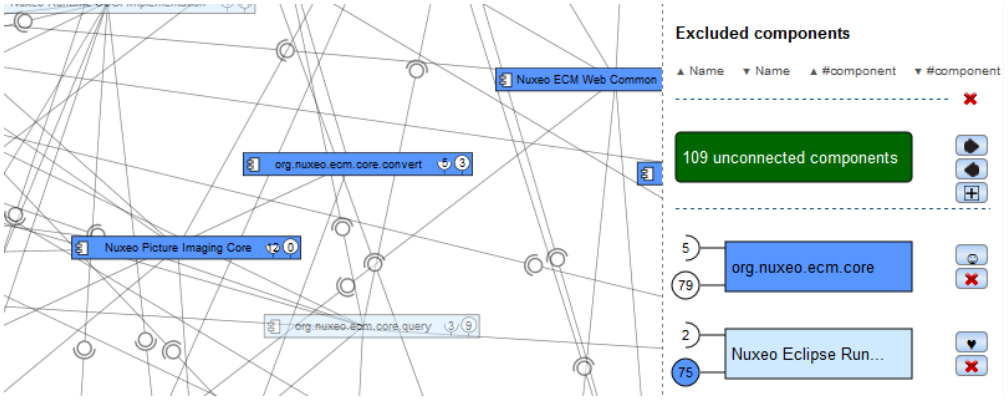


Figure 6.3: Excluded Components Connections Highlighting

For highlighting connections of more than one component Symbols and Delegates can be used. For several components from the SeCo area (those with

symbols' background highlighted by different than default blue colors) there are delegates shown in the diagram area as shown in Figure 6.4.

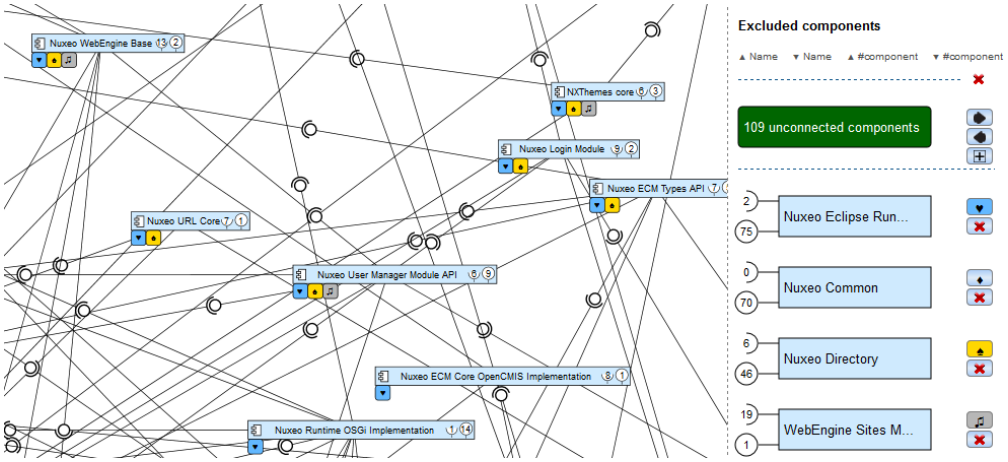


Figure 6.4: Using Symbols and Delegates

Each individual component shown in SeCo has its own button to remove it back to its original position in the diagram area.

6.1.3 Diagram Area Features Implementation

To ease the clarity when inspecting interfaces, the tool offers highlighting of a connection by a red colour and showing the interfaces involved in the connection, as shown in the green tooltip in Figure 6.5.

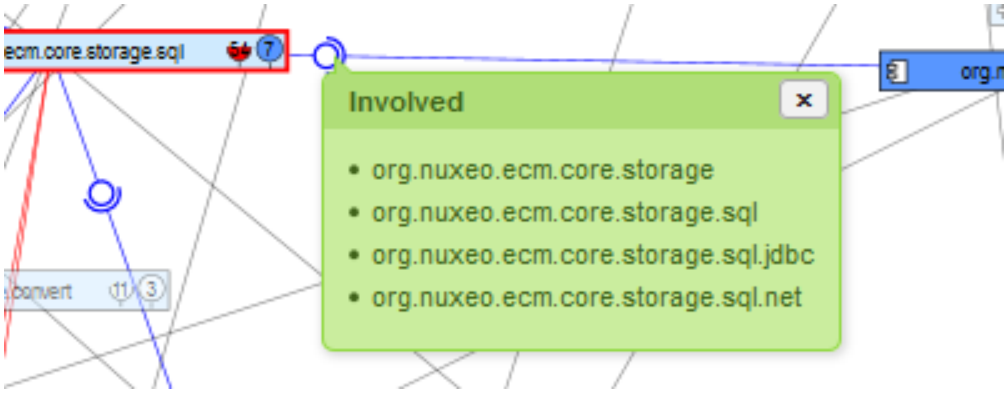


Figure 6.5: Clustered Interfaces Exploration

When a user clicks on a particular component, connected components are highlighted both in diagram area and SeCo. The colours correspond to the rules mentioned earlier as shown in Figure 6.6. Connections leading from/to connected components in the diagram area are also highlighted by corresponding colour. Components and connections, which are not highlighted, are less visible. It helps orientation and diagram simplification.

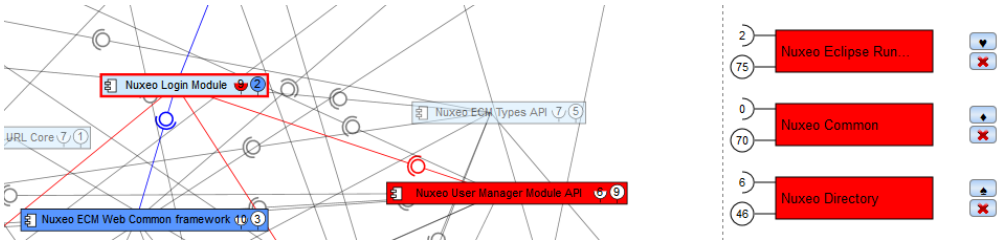


Figure 6.6: Connected Components Highlighting in Diagram Area

There is also a possibility to highlight the components connected only via required or only via provided interfaces. For this purpose there are respective symbols shown directly in a component displayed in the diagram area, as shown in Figure 6.7.

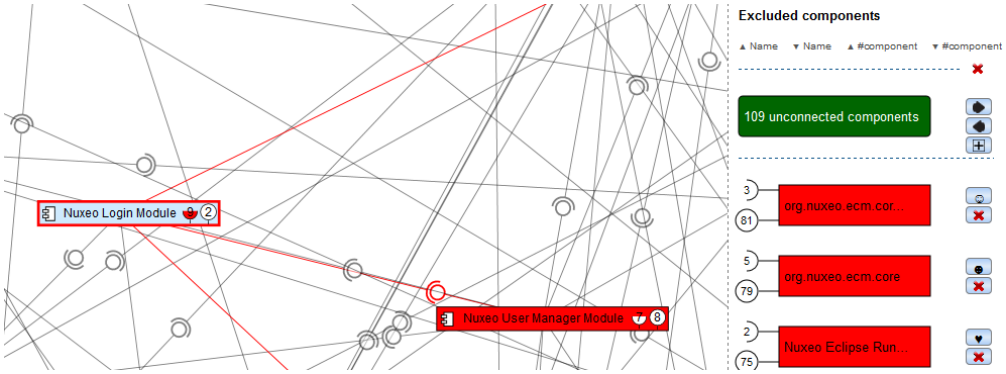


Figure 6.7: Required Interfaces Highlighting

6.1.4 Clusters Features Implementation

For forming clusters the concept of groups can be used. Any component from the diagram area can be added to an existing item (component or group) in SeCo. This is achievable by right-click action on a component in the diagram area, as shown in Figure 6.8.

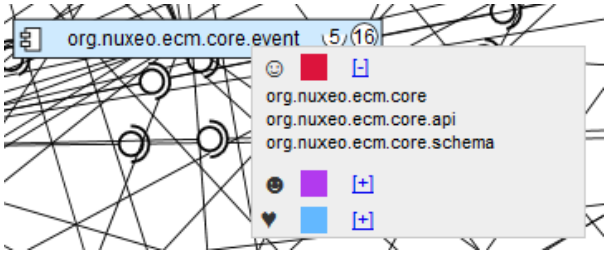


Figure 6.8: Adding Components from Diagram to SeCo Groups

The group in SeCo is represented by a list of components which it contains and one symbol (belonging to the whole group), as shown in Figure 6.9. Besides a symbol, any group can also have a name assigned. This helps create some semantical clusters better than using only symbols that are primarily for showing elided connection lines.

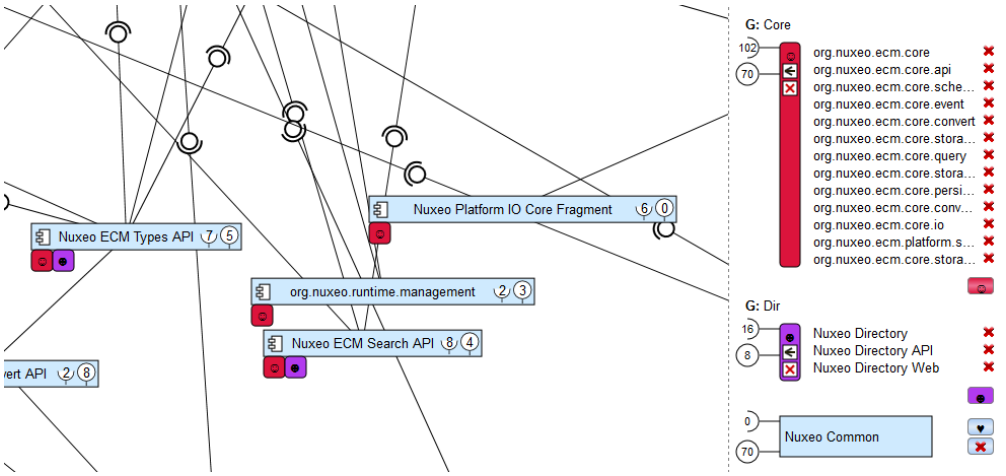


Figure 6.9: Forming Clusters with Group Feature

Clicking on this area also enables a user to highlight components connected to any of the components inside of this group, as shown in Figure 6.10 for group named Core. Connections of any particular component (from the group) can be also highlighted when a user clicks on its name, as shown in Figure 6.11 for `org.nuxeo.ecm.core.api` component.

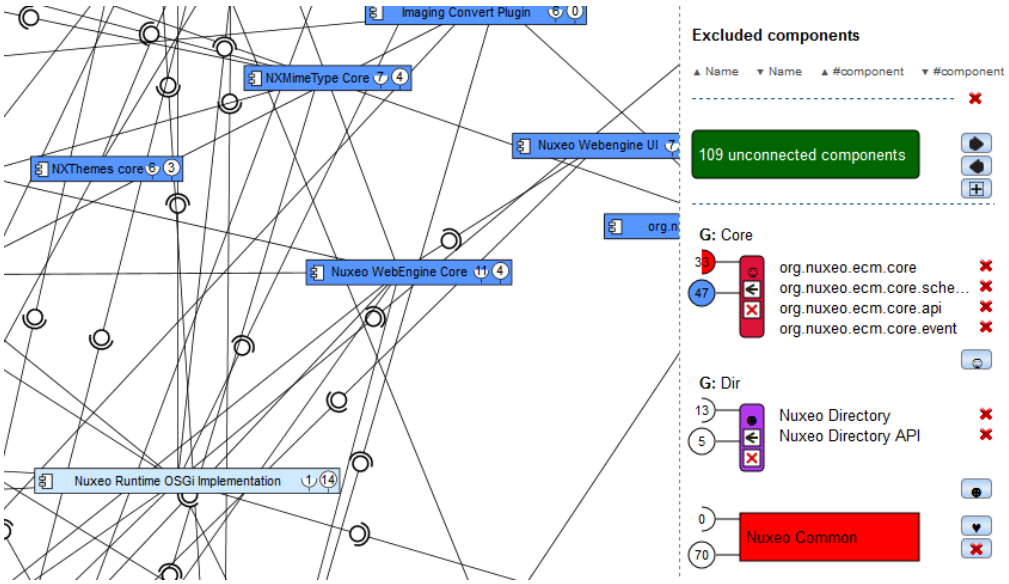


Figure 6.10: Highlighted Connections of a Group

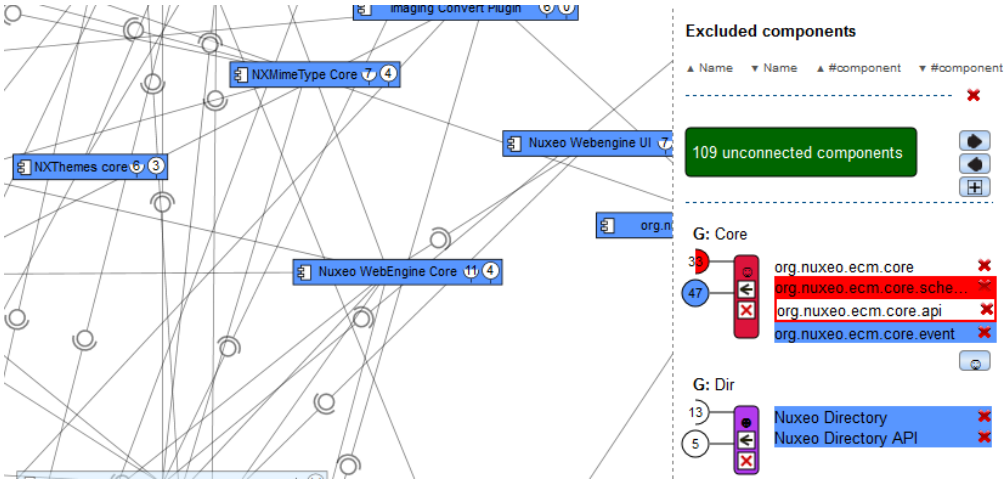


Figure 6.11: Highlighting Components Inside of a Group

When a logical cluster of components is formed there is a possibility to show the group which it represents as a Symbol (see Figure 6.12) in the diagram area. This allows a user to see the connections of this group, with the rest of the content in the diagram area, without any interaction. Compared to component visualization, we can see that there are additional icons in the Symbol's visual representation in the diagram area.

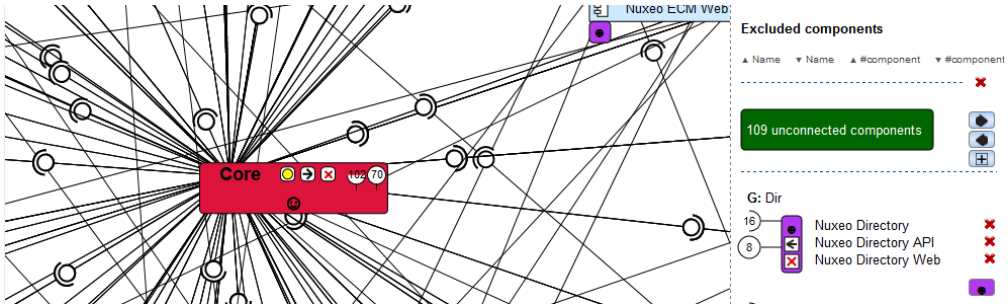


Figure 6.12: Showing Core Group as a Symbol

First one serves for expanding the group's graphical representation from a Symbol to a list of components. The list is shown in the diagram area inside a group box, as shown in Figure 6.13. Individual components inside the expanded list are equally interactive as when shown separately. It means a user can use highlighting of its required and provided interfaces. When list is shown, first icon in a group is changed to a underscore symbol. After clicking on this symbol the list is collapsed back to a symbol representation of the group. Second one (the arrow icon) moves the group from the diagram area back to the SeCo area. Third one (red cross icon) removes the group and shows individual components in the diagram area. The group shown in the diagram area can also use highlighting of required and provided components.

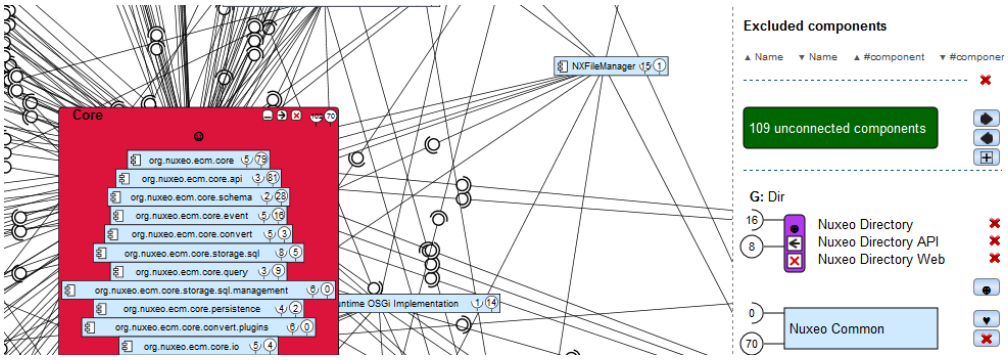


Figure 6.13: Group Expanded to a List of Components

6.1.5 Unconnected Component Feature Implementation

The Unconnected components item is implemented almost as designed in 5.5.6. There is a small difference in the look of the icons and there are not shown provided and required interfaces yet. Individual components can be moved to the diagram area. The CoCAEx application still remembers which components are unconnected and in case of need all of them can be moved to the Unconnected components item in SeCo by using the right arrow icon next to the green label (named Unconnected components in SeCo). When a user clicks on an unconnected component in the diagram area (exclude mode must be selected), a standard SeCo item is created for this component. One of the main purposes of this behaviour is that CoCAEx thus allows to create a group even for unconnected components.

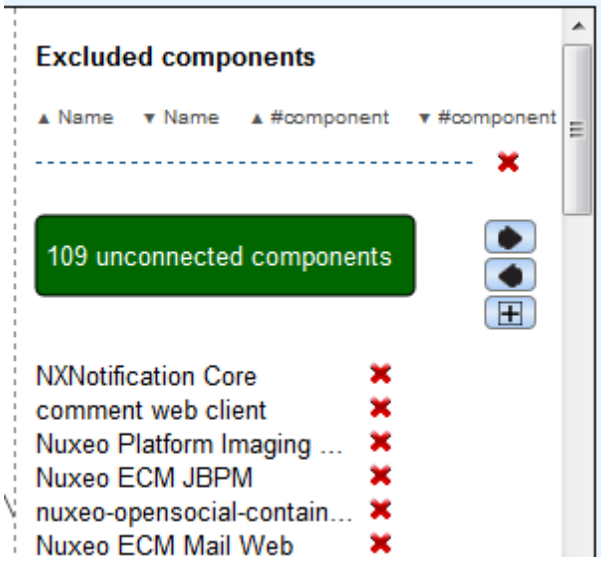


Figure 6.14: Unconnected Components in SeCo

Our preliminary experience with this tool shows that it is able to handle large diagrams without problems. We also discovered additional requirements for this tool that should be implemented to ease the work more. We describe these

requirements in Chapter 8.2.

6.1.6 Extra-functional Properties Visualization Implementation

This section describes the implementation of EFP visualization concepts mentioned in 5.8. Having CoCAEx application which enables to visualize diagrams, allowed us to show additional information in the diagrams. Part of our work focuses on EFP visualization, which was partially realized by EFFCC [63] tool. EFFCC is able to present EFP in a form of list of EFP for a component on the left side and corresponding EFP of the connected component on the right side, as shown in Figure 6.15.

Figure 6.15: EFFCC Tool [73]

CoCAEx application can receive the data from EFFCC and visualize the EFPs in a way described below.

There is a selection item in the CoCAEx's toolbar, where a user can select particular EFP from available ones, as shown in Figure 6.16. After such selection a diagram is refreshed. Exploration of clustered interfaces enriched by EFP was implemented in way shown in Figure 6.17.

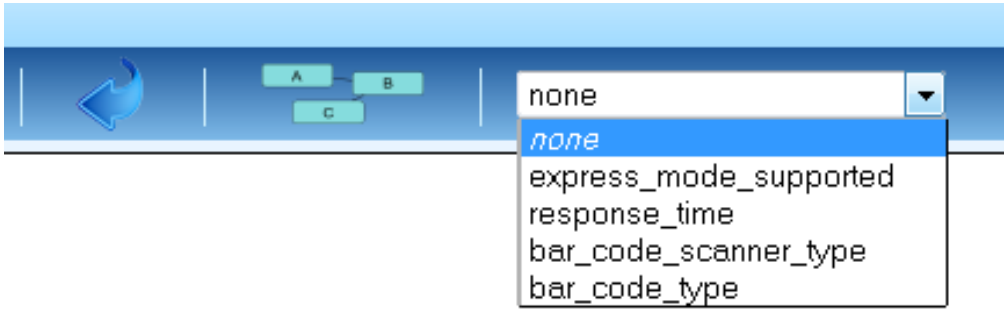


Figure 6.16: EFP Selection in CoCAEx Tool [73]

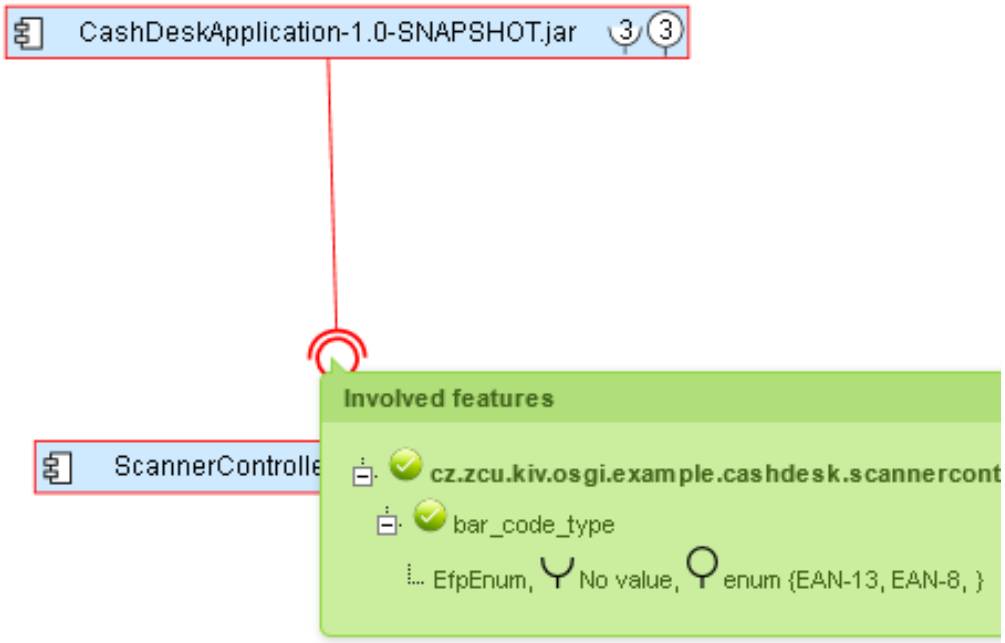


Figure 6.17: Exploration of Clustered Interfaces Enriched by EFP Implementation [73]

The scale of interface icon diameters is within range defined by `minInterfaceDiameter` (min) and `maxInterfaceDiameter` (max) values (in pixels, defined in `WEB-INF/web.xml` file). First mentioned is default value for interface connections which does not have any EFP defined. This range defines a full scale of possible sizes. The lowest (l) and highest (h) value of particular EFP define an existing range of values ($[l, h]$). Such range is than mapped to the scale (defined by $[min, max]$) according to Formula 6.1, so the $size$ to show is determined (in pixels).

$$size = \frac{e - l}{h - l} \cdot (max - min) + min \tag{6.1}$$

Where:
e is existing EFP value

In a case there are more EFPs defined in one clustered interface, the application currently counts an average from them for acquiring e value. To be able to visualize size that corresponds more precisely to EFP values inside one clustered interface, we would need to improve the data source (EFFCC). A function defining a distance metric between given EFP and the closest optimal value would be needed. Defining such general metric can be difficult considering the richness of possible EFP values (intervals, numbers, strings, characters, structured sets, etc.). Generally it would probably lead to a n-dimensional functions.

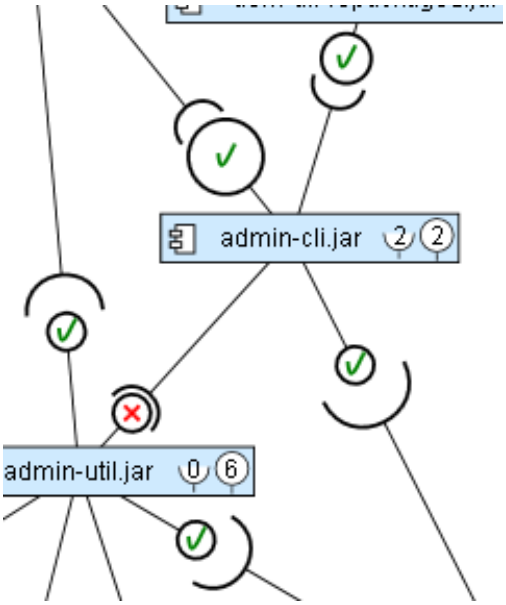


Figure 6.18: Interfaces Scaling According to Relative EFP Values Implementation [73]

6.1.7 Personalization and Publication

The reverse engineering process is generally a process of understanding the work and concepts created by someone else. A user doing a reverse engineering is usually able to understand given system to a certain level needed for particular task. There are also cases when a user wants to understand the whole system. For instance, for reimplementing it in other programming language. In both of these cases there is a risk of not understanding or overlooking something. If we consider that the reverse engineering work is usually done by multiple users independently in the world (who do not need to know about each other), we can assume following:

- Sharing a diagram about own system can speed-up the learning curve for other developers working with such system and does not cost much effort for the author.

- Diagram of an application shared by its author would likely help others to shorten the process creating such diagram by their own.
- Diagram of an application shared by another person would probably help others to shorten the process of creating such diagram (at least slightly).
- Multiple users working with the same system can get to know about each other. So giving a common web-based reverse engineering platform can help them find each other.

If the diagram from the authors exists, it is likely the most reliable source of information. On the other hand, if no diagram exists, users, doing the reverse engineering, can benefit from a wisdom of the crowd ([102]) and do not need to start the process from the very beginning. That is why we consider important to have the publication and sharing features available. It will also allow us to determine automatically one diagram that will consist of information extracted from diagrams (of the same application) of individual users.

When using the CoCAEx, the application shows the upload dialog first, as shown in Figure 6.19. Diagrams can be saved for later use and optionally marked as public. There are also predefined diagrams for demonstrative purposes.

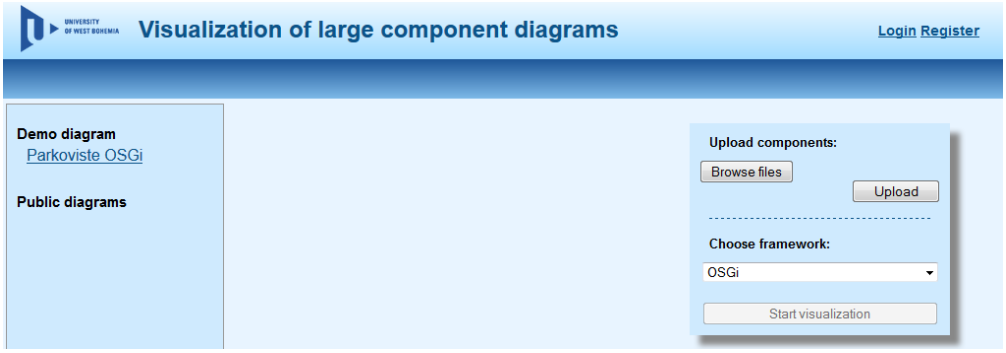


Figure 6.19: CoCAEx Application Upload Dialog

When uploading components, a user uploads the files to the server. The application then parses files of known formats and shows them in the list below the upload dialog, as shown in Figure 6.20. This list can be adjusted manually before the visualization is started. This can be done by pressing the Start visualization button. Unsupported data types are listed above the upload dialog.

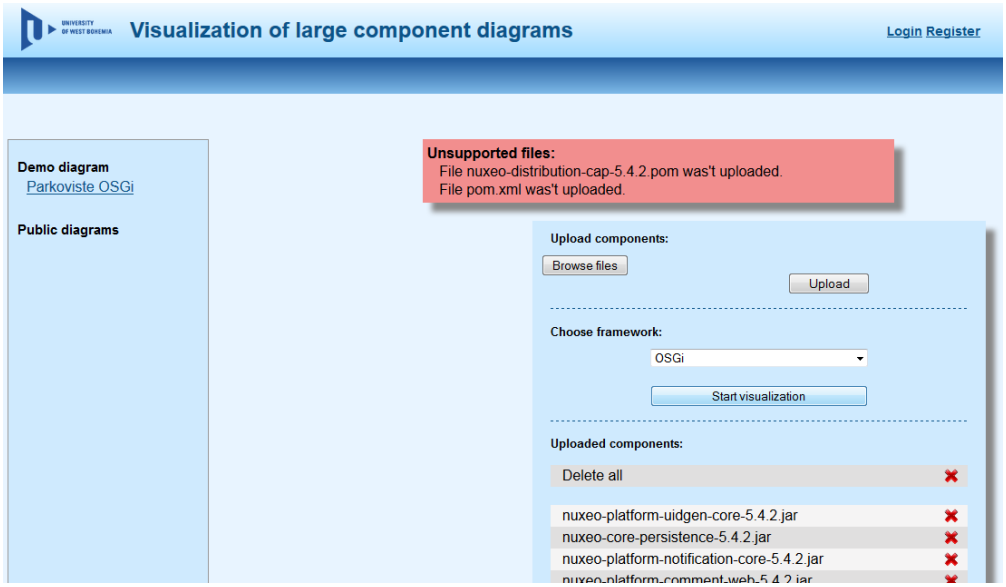


Figure 6.20: Uploaded Components to CoCAEx Application

The option of showing a Demo diagram shows a simple diagram where the application features can be quickly demonstrated without any need of uploading or logging in.

To offer a possibility of continuing the reverse engineering process later, CoCAEx offers diagrams saving. To be able to save a diagram a user needs to be registered. The registration requires name, email, login name and password to be filled.

When a registered user logs in, the application also allows diagram sharing. The idea behind the sharing concept is described below.

The upload dialog for users who are logged in is shown in Figure 6.21.

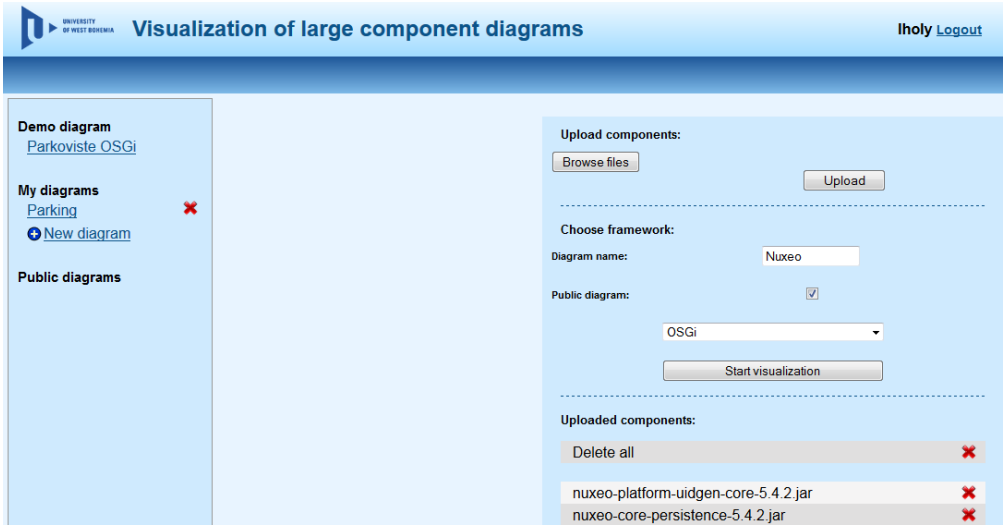


Figure 6.21: Public Diagram Creation Dialog

If a diagram is named, it is possible to save it. It means that positions of all components in the diagram area are stored in the database for later loading. Currently the application does not remember state of other features (highlighting, SeCo content, etc.). For named diagrams there is the save (diskette) icon shown in the toolbar as illustrated in Figure 6.22. If a diagram is not named it is considered as a temporary work or test of the application and there is no possibility to save it.

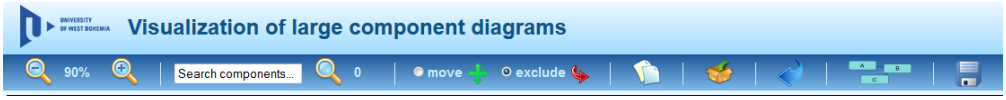


Figure 6.22: Save Icon for Named Diagrams

Each diagram can be marked as public and thus shared to all users, even to the not registered ones. The public diagrams are read-only for all users, except the owner. Users, who are logged in, can copy diagrams of other users to continue working with their own copy, as shown for `demouser` in Figure 6.23.

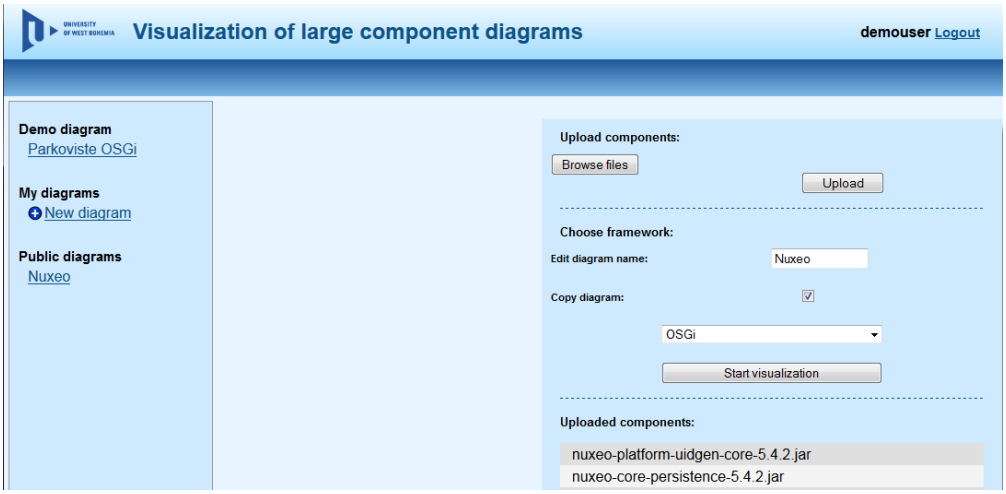


Figure 6.23: Copying Public Diagram

6.1.8 Application Features Overview

The application features are described in following list. They can be divided into four main categories:

I. global features,

   (a) removing nodes with the highest degree to the SeCo area,
   (b) searching and highlighting components in the diagram,
   (c) modes toggling,
   (d) component selection refinement,

II. SeCo features,

(a) items creation, sorting, deletion,
(b) unconnected components list,
(c) item's symbols using,
(d) groups using,

III. diagram area features,

(a) delegates of connected components showing,
(b) scrolling and zooming in diagram area,
(c) automated nodes layout (after first load),
(d) automated nodes layout on demand,
(e) manual layout adjusting,
(f) connected elements highlighting,
(g) visual suppressing unfocused components,
(h) provided and required interfaces highlighting,
(i) clustered interfaces exploration,
(j) group as symbol representation,
(k) group as list representation,
(l) adding components from diagram to SeCo groups,

IV. personalization and publication,

(a) login and registration,
(b) saving diagrams,
(c) sharing diagrams,
(d) demo diagrams.

Coverage of general design concepts (mentioned in Section 5.3) by features is shown in Table 6.1. Beside mentioned table, all the features should support reverse engineering process improvement. The set of features related to personalization and publication (IV. part mentioned above) should improve long-term usability and shorten the initial learning curve of CoCAEx. Coverage of main problems (mentioned in Section 1.2) by features is shown in Table 6.2.

6.2 Technologies Selection

For implementing above described techniques a graph framework can be used, because the component diagram can be considered as a graph. There are many available graph visualization frameworks (commercial or free to use). Their list for Java can be found at [4]. We focused on framework's ability to interact in a short time with the user while displaying large amount of elements. The other important abilities were available documentation, graph layouts, customizability and size of the community around framework.

Feature from list in Sec. 6.1	Overview	Zoom and Filter	Details on demand and Relate	History	Extract
I. (a)		*			
I. (b)		*	*		
I. (c)		*			
I. (d)				*	
II. (a)		*		*	
II. (b)		*			
II. (c)			*		
II. (d)		*	*		
III. (a)			*		
III. (b)		*			
III. (c)	*				
III. (d)	*				
III. (e)		*			
III. (f)			*		
III. (g)		*			
III. (h)			*		
III. (i)			*		
III. (j)			*		
III. (k)			*		
III. (l)		*			
IV. (a)					*
IV. (b)					*
IV. (c)					*
IV. (d)					*

Table 6.1: Feature-Concepts Coverage

Feature from list in Sec. 6.1	Too big to keep orientation	Trace "long" connections	Visual clutter reduction
I. (a)	*		*
I. (b)	*		*
I. (c)			
I. (d)	*		*
II. (a)	*		*
II. (b)	*		*
II. (c)	*	*	*
II. (d)	*	*	*
III. (a)	*	*	*
III. (b)	*	*	
III. (c)	*		*
III. (d)	*		*
III. (e)	*		
III. (f)	*	*	
III. (g)	*	*	*
III. (h)	*	*	
III. (i)	*		*
III. (j)	*		*
III. (k)	*		*
III. (l)	*		*
IV. (a)			
IV. (b)			
IV. (c)	*		
IV. (d)			

Table 6.2: Feature-Problems Coverage

At first we chose JUNG [3], JGraph[1] and Zest[2] frameworks as suitable for visualization of large graphs for further comparison. These were chosen because of our knowledge of Java. Mentioned frameworks are able to work with both directed and undirected graphs. They also provide GUI for work with the displayed graph and layout functions. After testing these three frameworks we decided for JUNG. This framework is described in following section.

6.2.1 JUNG Framework

The JUNG [3] stands for Java Universal Network/Graph Framework. It is a software library written in Java and compatible with Swing[3]. It supports both directed and undirected graphs as well as hypergraphs. It allows users to annotate graphs, entities, and relations with metadata. It is also possible to use Java applets as shown in Figure 6.24.

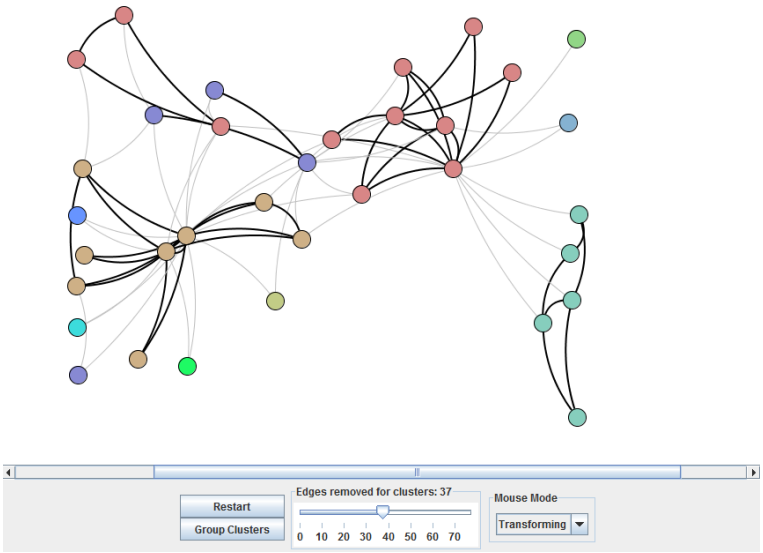


Figure 6.24: Example of JUNG Applet Showing both Clustering and Layout (Fruchterman-Reingold) Algorithm

We chose the JUNG framework as the most suitable for our needs, because of good documentation and overall functions. Showing 5000 nodes interconnected with 5000 edges last 28 seconds in JUNG framework whereas it takes 1180 seconds in JGraph framework.

We implemented a prototype showing the large component diagram. We have customized the connections lines to be shown as connected interfaces. Although this framework is very good in graph visualization, we considered the implementation of described techniques problematic, mainly due to large amount of specific customizations. Thus we decided not to use any framework and implement the desired techniques by using basic drawing primitives.

[1] http://www.jgraph.com/
[2] http://www.eclipse.org/gef/zest/
[3] http://docs.oracle.com/javase/6/docs/technotes/guides/swing/

6.2.2 HTML5 and Java EE

Based on the experience with frameworks in Java, we decided not to use one. We preferred to use only basic drawing elements because they provide fast prototyping of our novel techniques. Considering this decision, it was not essential to use Java SE and we were able to use web-based technologies as well. At the beginning we decided for Java SE, mainly because of our knowledge of this edition of Java language. But in case of using only basic drawing primitives and elements, we were able to implement such visualization as a web application even with our limited knowledge of client side web technologies (mainly JavaScript[4,5]). This approach is better for faster and easier sharing of our techniques to broader audience directly on the Internet. It can be also used as a tool for reverse engineering for anyone without necessity of installation.

As a backend technology we use servlets from Java EE technology, mainly because of Java implementation of ComAV tool (see Section 6.3). For frontend we use modern technologies such as HTML5, JavaScript, jQuery[6] framework and CSS3. These technologies are well-known and provide desired features seamlessly integrated in the web page, whereas JUNG uses applets which cannot be connected to the rest of the web page so easily as the previously mentioned technologies. We use `canvas` and SVG elements from HTML5 to represent the nodes of the diagram. Although HTML5 technology was not fully supported[7] by all main browsers, it provides uploading of multiple files, which is used for uploading components. Also desired features such as SVG support or Canvas are likely to be stable in the future.

An important factor for work with large diagrams is an application response time. The longest delay in the reverse engineering process is the initial upload and the diagram creation. It is caused by the fact that applications can have hundreds of megabytes and thus their uploading and processing can take some time, depending on the network connection speed and computer performance. With above mentioned technologies, we designed the application to load all necessary data at the beginning (during initial load) and thus eliminate further delays during work. We verified that even for application having around one thousand components, the application works without uncomfortable delays after initial load. We loaded a diagram containing 914 components, which have in total 113 MB and took to show 10 seconds[8].

[4] https://developer.mozilla.org/en-US/docs/Web/JavaScript

[5] http://www.ecmascript.org/

[6] http://jquery.com/

[7] Compatibility can be checked at http://caniuse.com/

[8] System configuration: Intel(R) Core(TM) i5-2300 @ 2.80 Ghz, Windows 7 64-bits, Firefox browser

6.3 Component Application Visualizer

We implemented most of above mentioned techniques as the web application. It uses ComAV tool [99] as a data source. ComAV is a versatile an extendible platform for visualization and reverse engineering of component-based applications.

It offers the possibility to use multiple component models (currently OSGi, EJB 3 and SOFA 2 are supported) and different visualization styles. It uses component-model independent data format to store a reverse engineered structure of component-based applications and as an input for any visualization plug-in. Its architecture is illustrated in Figure 6.25.

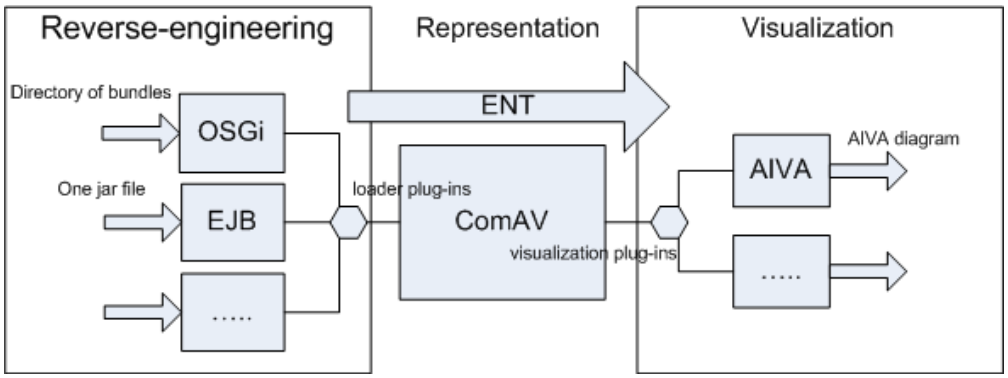


Figure 6.25: Architecture of ComAV Tool [99]

6.4 CoCAEx Application Internal Data Flow

The CoCAEx tool is able to load and visualize the components reverse-engineered by the ComAV tool. A user first picks component on a local machine and uploads them to the server. The ComAV tool creates the model of the application and the CoCAEx tool shows the application diagram in the webpage.

Figure 6.26 shows a data flow between ComAV and CoCAEx and CoCAEx's client and server parts. While CoCAEx server emits raw component data to ComAV and receives reverse-engineered models, the CoCAEx client provides model's visualization and user interaction. Reverse-engineered models are sent from the server to a web browser client in JSON format[9].

[9] http://json.org/

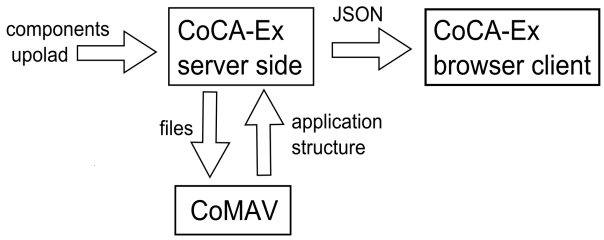


Figure 6.26: CoCAEx Architecture

Chapter 7

Evaluation of the Proposed Approach

This chapter is based on a paper submitted to publication in Journal of Visual Languages and Computing.

In order to find out to what degree of interactivity is useful or which techniques used in CoCAEx approach are most beneficial for shortening the time needed for exploring the structure of complex component-based applications, we performed a controlled user study. This section provides the details of its goal and mechanics, while the following sections describe and discuss the results.

It discusses the state-of-the-art standard UML tool approach as a baseline and our new approach described in this thesis. While our approach reduces visual clutter, this comes at the cost of hiding information about the actual interfaces used between the components. We overcome this problem via provided interactivity.

7.1 Baseline Approach

Typical features for state-of-the-art approaches are scrolling, panning, zooming, search of filtering. While it makes navigation better than on the paper, it still relies on the statical nature of a diagram keeping its original pros and cons. Therefore, one of the state-of-the-art UML tools should be selected to objectively investigate its usability; this is not an easy task as there are a lot of tools with large (and not fully comparable) sets of features.

For the evaluation of our approach, from the currently most used tools (such as Enterprise Architect[5], MagicDraw[7], IBM Rational Software Architect[9], Visual Paragigm[10]) we chose IBM Rational Software Architect (RSA). The reason for selecting RSA is that it can be considered as the most advanced, industry-strength tool with a lot of additional features and widely used by the biggest software houses. In other words, we decided to choose the most challenging competitor to compare with CoCAEx.

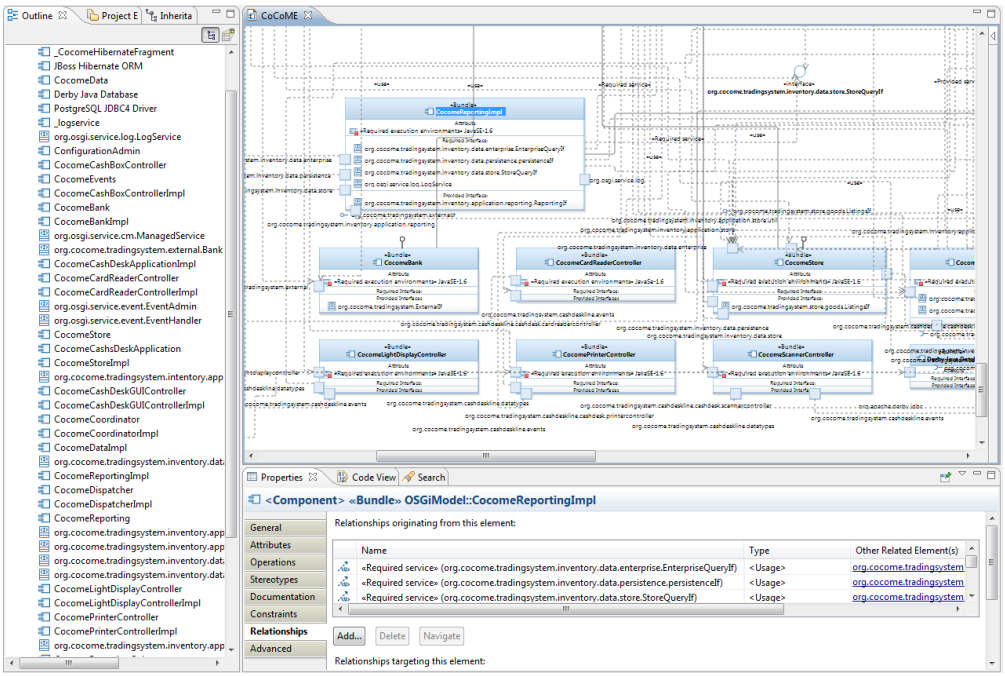


Figure 7.1: An Outline and Properties View of IBM Rational Software Architect

Besides all standard features included in other tools, RSA supports some advanced ones that allow users to manipulate the diagram, like changing the layout of nodes, changing the line routing and modifying the look of components and interfaces. Another added value is in its "*properties view*" displayed at the bottom of the screen; see Figure 7.1. This view shows all the details about components and relations and, most importantly, can be used to navigate to related components. For example, the "*Relationships*" tab shows a list of all elements that use or are used by a given component. This list clearly specifies which kind of relation is used and which components are related. Each relation line contains a link which can be used for acquiring more information about the line.

7.2 User Study

This section provides the details about the goal and mechanics of the performed user study.

7.2.1 Goal of the Study

This study evaluates whether it is faster to analyse the structure of component-based applications interactively with the CoCAEx notation and tool rather than to study the structure with one of the good state-of-the-art UML tool, in particular RSA. Our *null hypothesis* is that the user performance is approximately the same. This knowledge of the effects of the approach is important because the level of interactivity used in CoCAEx is high and could negatively affect the

user's performance while he/she collects multiple detailed information, specifically because a lot of this information is hidden and revealing it requires user interaction.

In the study we are using two different tools for experiments. Ideally, our notation as well as standard UML notation would be tested in one tool to use exactly the same look and feel for both notations as well as the same basic framework of user interaction. However, this was not possible as CoCAEx cannot be switched to standard UML notation and our notation is obviously not implemented in RSA. Integration of one of the notations to the opposite tool is technically difficult. For that reason, we decided to perform the study with two tools and mitigate the possible problems caused by their differences by a careful design of the experiment procedure.

The study simulated the activities performed during one step of architecture analysis. These activities are focused on collecting knowledge about components' features, dependencies and overall context consisting of related components. The concrete set of tasks used in the study, which was based on those activities, is discussed thoroughly further below.

7.2.2 Participants

Twelve participants were recruited from two different universities. All participants were young software engineers and all were academics or Ph.D. candidates (the use of academics and Ph.D. candidates was encouraged by Sensalire et al. [94]). All were proficient users of computers and had sufficient knowledge of UML to fully understand the presented diagrams and they had also proficiency in analysis of component-based structures and applications.

7.2.3 Apparatus

The hardware used in the study consisted of standard PC (Intel Core i5 at 2.8 Ghz with 8GB RAM), 24 inches LCD display (resolution 1920×1080 pixels), PC keyboard and optical mouse with 2 buttons, running Windows 7. Participants used RSA (version 8.0.4) and CoCAEx tools (version 0.3) in the study. Both tools were running smoothly on the selected hardware.

During the study, users were analysing the Common Component Modeling Example application (CoCoME) [91] – an information system for supermarket chains developed originally with the aim of comparing different approaches to component-based software modelling. The CoCoME application, which represents a medium-size application, consists of about 40 components divided into three main parts. First is a cash desk, including barcode scanners, credit card readers, etc. The second part is a store back office server and a store client. Finally, the chain part consists of an enterprise server and client applications. For the purposes of the test we have used our own implementation of CoCoME (available at [8]) in the OSGi component model [86] – the diagrams in Figures

7.2 and 5.5 visualize the structure of this CoCoME implementation in CoCAEx and UML, respectively.

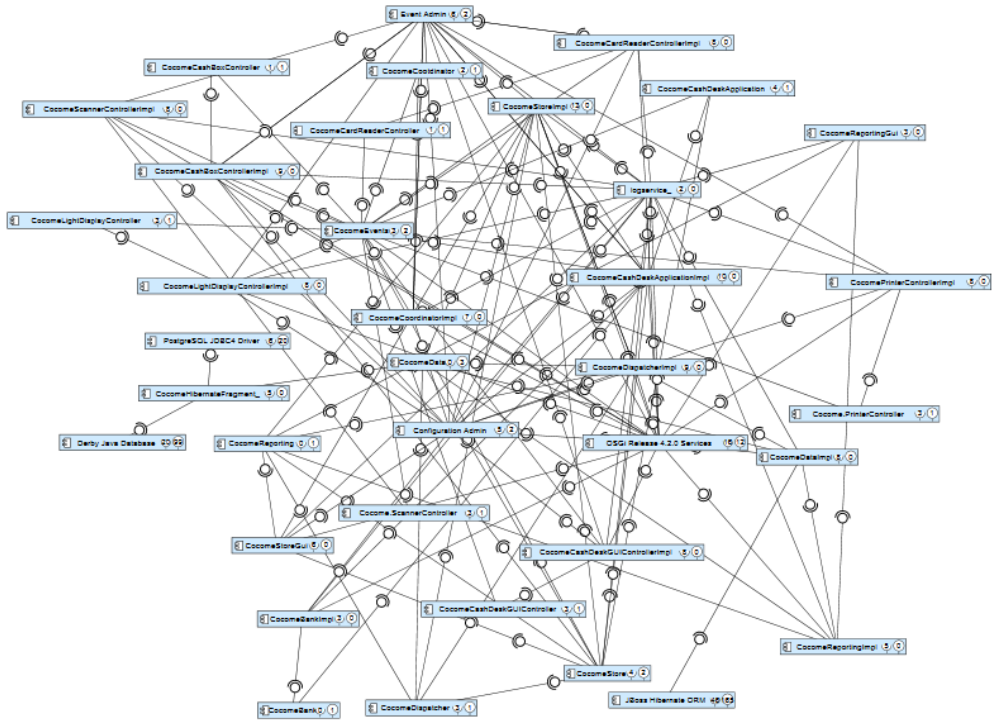


Figure 7.2: CoCoME Application Shown in CoCAEx

The UML diagram (in RSA) of the CoCoME application can be seen in Figure 5.5. There are components containing names, additional details and large number of non-clustered connections. If we compare this Figure to CoCoME shown in CoCAEx application (in Figure 7.2), we can see that interface clustering used in CoCAEx reduced the number of connections. Also details of components hiding saves space in the diagram area.

7.2.4 Design

The study was organized as one factor (with two levels) within-subject design. The independent variable was the used analytical tool. The order of tools was counter balanced and the group effect (asymmetrical transfer of skills from tool one to tool two) was evaluated.

The main measure was speed, measured as a number of seconds needed to accomplish each task.

7.2.5 Procedure

The test was performed at two locations with the same procedure. At each location, the experiment was performed in a dedicated room where participants were not disturbed. Before the experiment was started, participants adjusted

the position of the display and the mouse to feel comfortable.

The moderator of the experiment first explained the user interface of the first tool. The experiment began with a training session. In the training session participants were asked to accomplish five tasks very similar to those they will perform in the actual experiment, but using visualized structure of different small-size application. During the training session, the moderator helped participants to accomplish all tasks if necessary. The goal was to let participants get familiar with the first approach, get used to the experiment procedure and minimize any learning effects. Specifically, we focused on minimization of impact of different complexity of tools on the results of the study. Namely, in case of RSA, we gave hints to users of what features they will need in the study. This should prevent a situation when the user gets lost in wide amount of features provided by RSA. The training lasted 15-20 minutes and it ended when the participant felt confident and familiar with both tools and able to perform all types of tasks used in this user study.

Training was followed by the experiment session with the first tool. During the experiment session, participants were asked to proceed as quickly and accurately as possible. Between each task, participants were allowed to take a short break. After the sessions with the first tool the same procedure was repeated for the second tool. The whole study lasted about 1 hour.

Participants accomplished five tasks. These were given to participants as follows:

T1 – Which components use interfaces provided by CocomeDataImpl? The task was focused on analysis of the parts of the system which will be influenced if some particular component is changed.

T2 – Which components are not from CoCoME core (are third party)? The task was focused on analysis of the system structure, mainly discovery of the core of the system and usage of the third party components in the system.

T3 – Which packages need CocomeDataImpl from CocomeData? The task was focused on analysis of the relation between two components in the system.

T4 – Which components do not require or provide interfaces to any other components (are unconnected)? The task was focused on analysis of unconnected components that are suspicious, because they are probably using some non-standard way of communication with other components.

T5 – Which components require or provide interfaces to any of CashDesk components in CoCoME? The task was focused on analysis how a particular part (usually feature) of the system is connected with the rest of the system.

The tasks were defined based on our experience with the structure of component-based applications and based on hints obtained during interviews with several software engineers from local software companies. Individual tasks are typical tasks used iteratively in global task that deals with the question of what is the structure of application and how are particular components integrated in the CoCoME application. One has to find out what these components offer and require and uncover their ties to other components, simulating the activities performed during one step of the architecture analysis.

Two questionnaires were given to participants during the test. At the beginning of the test, participants were asked about the experience with UML diagrams and UML editors. After the data collection, participants completed a questionnaire investigating their subjective judgment about the used approaches.

7.3 Results and Discussion

This section provides detailed results of the study for each approach and their comparison. As the reader may notice, the results differ greatly depending on the participant. This was caused by individual perception, and orientation abilities. A lot of attention was paid to preparing all participants thoroughly, see Section 7.2.5. Task completion times of all participants for each task and basic statistics of individual tasks are presented in Tables 7.1, 7.2, 7.3, 7.4. Comparison of mean times of individual tasks is in Figure 7.3. Detailed comparison of average, median, minimal and maximal times is in Figure 7.4.

For statistical analysis of the results we have used the *repeated measures ANOVA* (ANalysis Of VAriance) [74]. We report the results in F-statistics notation $F_{X,N} = V, p < \alpha$ where X is number of factors considered in the study, N is number of participants in between-subject design (note that correction is applied in our case due to the within-subject design). If value $V \leq 1$ then it is impossible for the means of the tested approaches to be significantly different. The amount V rising above 1 is an indication of the size of the difference in the means of the tested approaches. If the *null hypothesis* is true then p is the probability of obtaining the observed data. In other words, if the value of p is lower than confidence level α (α is typically 0.05) then the difference in the means of the tested approaches are unlikely to occur in the view of the *null hypothesis*.

From this follows that we can reject the null hypothesis when $V > 1$ and $p < 0.05$. When $V \leq 1$ or $p \geq 0.05$ we report that the difference in means of the tested approaches is *not significant* (shortened to "ns"), in such case the *null hypothesis* cannot be rejected.

In the following subsections we evaluate the *null hypothesis* separately for each task. Later, in the summary, we evaluate the *null hypothesis* for all tasks together.

7.3.1 Task T1 – Which components use interfaces provided by CocomeData-Impl?

There was a significant difference in speed between RSA and CoCAEx ($F_{1,11} = 5.758, p < .05$). The average speed for RSA was 67s and for CoCAEx 29.75s ($2,25\times$ faster than RSA). The group effect was not detected.

In both cases, the users searched for the component and then they analysed its provided interfaces. For RSA, the analysis of provided interfaces took longer time, while for CoCAEx this information was immediately visible, when the component was selected.

7.3.2 Task T2 – Which components are not from CoCoME core (are third party)?

There was no significant difference in speed between RSA and CoCAEx ($F_{1,11} = 2.327, ns$). The average speed for RSA was 32.25s ($1,34\times$ faster than CoCAEx) and for CoCAEx 43.17s.

This was the only task where RSA was faster than CoCAEx, but not significantly. With CoCAEx, the users were searching for the components in the graph, which took them additional time to skim through the graph. On the other hand, with RSA, the users searched for the components in the list in Outline window, cf. Figure 7.1, which was very quick.

7.3.3 Task T3 – Which packages need CocomeDataImpl from CocomeData?

There was a significant difference in speed between RSA and the CoCAEx ($F_{1,11} = 12.954, p < .05$). The average speed for RSA was 97.83s and for CoCAEx 25.17s ($3.89\times$ faster than RSA). The group effect was not detected.

In this task CoCAEx allowed fast analysis of imported/exported packages between two components, which is not immediately visible in the graph, but it is shown as popup when mouse is over the interconnection between two components. In RSA, participants usually used the project tree and compared full lists of exported/imported packages.

7.3.4 Task T4 – Which components do not require or provide interfaces to any other components (are unconnected)?

There was a significant difference in speed between RSA and CoCAEx ($F_{1,11} = 14.905, p < .05$). The average speed for RSA was 121.83s and for CoCAEx 8.58s ($14.20\times$ faster than RSA). The group effect was not detected.

In this task the difference in performance between RSA and CoCAEx was the biggest. This was due to the fact that with RSA the participants had to find

Participant	T1	T2	T3	T4	T5	All
P1	00:33	00:39	01:34	01:07	03:47	07:40
P2	02:03	00:30	01:45	01:38	04:55	10:51
P3	01:05	00:30	01:18	01:47	03:24	08:04
P4	01:09	00:32	00:28	03:02	03:23	08:34
P5	03:35	00:30	04:59	07:06	04:37	20:47
P6	00:54	00:26	00:21	01:02	03:26	06:09
P7	00:26	00:32	00:27	01:53	03:22	06:40
P8	00:50	00:32	01:57	00:55	03:36	07:50
P9	00:45	00:49	02:15	01:43	03:52	09:24
P10	00:57	00:27	01:31	01:17	03:31	07:43
P11	00:40	00:23	01:34	01:41	04:15	08:33
P12	00:27	00:37	01:25	01:11	03:40	07:20

Table 7.1: Time [min:sec] Measured for Each Participant and Each Task in RSA

out the unconnected components manually, however CoCAEx allowed semiautomatic extraction and listing of all unconnected components in the panel on the right.

7.3.5 Task T5 – Which components require or provide interfaces to any of CashDesk components in CoCoME?

There was a significant difference in speed between RSA and CoCAEx ($F_{1,11} = 516.403, p < .05$). The average speed for RSA was 229s and for CoCAEx 69.08s (3.31× faster than RSA). The group effect was not detected.

This was the most complex task which focused on searching of multiple CashDesk components and analysis of interface connections between these components and rest of the components in the CoCoME application. With RSA, this task required searching of CashDesk components in the graph and iteration through their provided/required interfaces. With CoCAEx, there were two approaches how to achieve this task, both of them took similar time. In the first case, the participants analysed CashDesk components' interfaces in the graph, similarly as in the Task 1. Alternatively, participants extracted CashDesk components to the SeCo area (right sidebar) into one group or individually, and then highlighted all components connected through required/provided interfaces.

7.3.6 Subjective Evaluation

We also asked participants about the orientation in the visualized structure of the CoCoME application, the level of comfort while working with the tools, and any other suggestions.

All participants described the orientation in structure of the CoCoME applica-

Participant	**T1**	**T2**	**T3**	**T4**	**T5**	**All**
P1	00:23	00:29	01:00	00:30	01:06	03:28
P2	00:26	00:41	00:31	00:12	02:11	04:01
P3	00:23	00:34	00:28	00:05	01:39	03:09
P4	00:16	00:46	00:11	00:05	00:58	02:16
P5	00:41	01:51	00:32	00:11	01:52	05:07
P6	00:25	00:55	00:15	00:05	00:39	02:19
P7	00:46	00:29	00:22	00:10	00:47	02:34
P8	00:22	00:49	00:23	00:05	00:57	02:36
P9	00:38	00:50	00:26	00:05	00:58	02:57
P10	00:14	00:23	00:21	00:06	00:54	01:58
P11	00:53	00:18	00:13	00:04	00:41	02:09
P12	00:30	00:33	00:20	00:05	01:07	02:35

Table 7.2: Time [min:sec] Measured for Each Participant and Each Task in CoCAEx.

Measure	**T1**	**T2**	**T3**	**T4**	**T5**	**All**
Mean	01:07	00:32	01:38	02:02	03:49	09:08
Median	00:52	00:31	01:32	01:39	03:38	07:57
Min	00:26	00:23	00:21	00:55	03:22	06:09
Max	03:35	00:49	04:59	07:06	04:55	20:47
Std dev.	00:51	00:07	01:10	01:37	00:30	03:42

Table 7.3: Sum of Times for all Tasks and Statistical Measures in RSA.

Measure	**T1**	**T2**	**T3**	**T4**	**T5**	**All**
Mean	00:30	00:43	00:25	00:09	01:09	02:56
Median	00:26	00:38	00:23	00:05	00:58	02:36
Min	00:14	00:18	00:11	00:04	00:39	01:58
Max	00:53	01:51	01:00	00:30	02:11	05:07
Std dev.	00:12	00:23	00:12	00:07	00:28	00:52

Table 7.4: Sum of Times for All Tasks and Statistical Measures CoCAEx.

tion visualized in our notation as better. Further, all participants stated that it was more comfortable to solve all tasks in CoCAEx than in RSA. Four participants stated that the visual highlighting of the components related to the selected component significantly improved their orientation in the structure of the application. Three participants suggested to allow the selection of more than one component and visually highlight components related to all selected components. Two participants would like to use the same approach as used in CoCAEx for components also for classes and class diagrams.

7.3.7 Observation

The measured times needed to accomplish each task by each participant demonstrate that the tasks T1, T3, T4 and T5 were accomplished on average significantly faster in the CoCAEx tool than in RSA. The only exception is the task T2, which the participants accomplished on average faster with RSA. Overall speed comparison showed that the tasks with the CoCAEx tool were accomplished on average 3× faster then with RSA; see Figure 7.3. In the majority of tasks, the maximum values in CoCAEx were lower than the median values for RSA; see Figure 7.4. These results show that even the slower CoCAEx users were faster than at least half of the users in RSA for majority of tasks. The median values are averagely placed at first fifth of the minimum-maximum range.

The results indicate that the structure of the CoCoME application visualized in our notation is less cluttered which leads to faster orientation in the structure of the application. Further, the visual highlighting and interaction features of components related to the selected component provided by CoCAEx allowed participants to visually detect the related components much faster.

From these facts, we can conclude that the abilities to interact with the visualized structure of component-based software systems and to provide visual cues to ease identification of related components is better with CoCAEx. There was a significant difference in speed between RSA and CoCAEx for summed accomplished task times ($F_{1,11} = 46.581, p < .05$). Results for individual and summed accomplished task times allows us to reject the *null hypothesis*, i.e. that both tools provide the same speed of analysis of component-based application.

From the subjective point of view the participants perceived the orientation in our notation as better in comparison to the standard UML notation. They also subjectively perceived comfort of work in CoCAEx as better in comparison to RSA. All participants also answered that CoCAEx provides a clearer diagram that is more readable and understandable. They mentioned two main reasons: highlighting and simplicity of use.

There is a combination of two factors influencing participants' performance in each task, interactivity of tools and visual notation. This fact means that we cannot exactly separate impact of interactivity and the notation. As a consequence, we cannot be sure whether better performance in certain tasks

was caused more by the interactivity or by the notation. For instance, we could think that a task was performed faster due to an interactive feature (e.g. usage of interface clustering) while in fact a better colour encoding or contrast helped a user more.

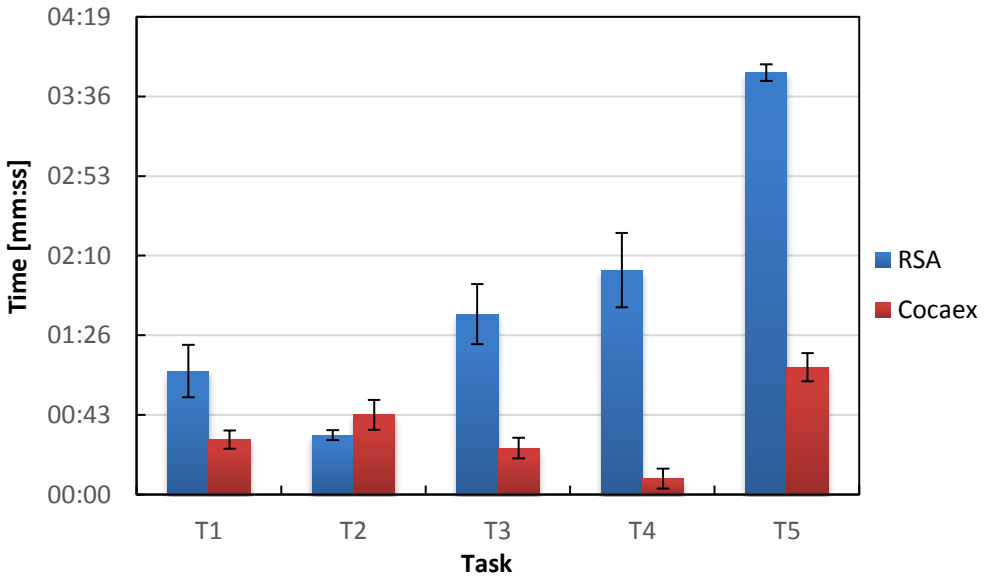


Figure 7.3: Comparison of Average Times Needed to Accomplish the Tasks in RSA and in CoCAEx

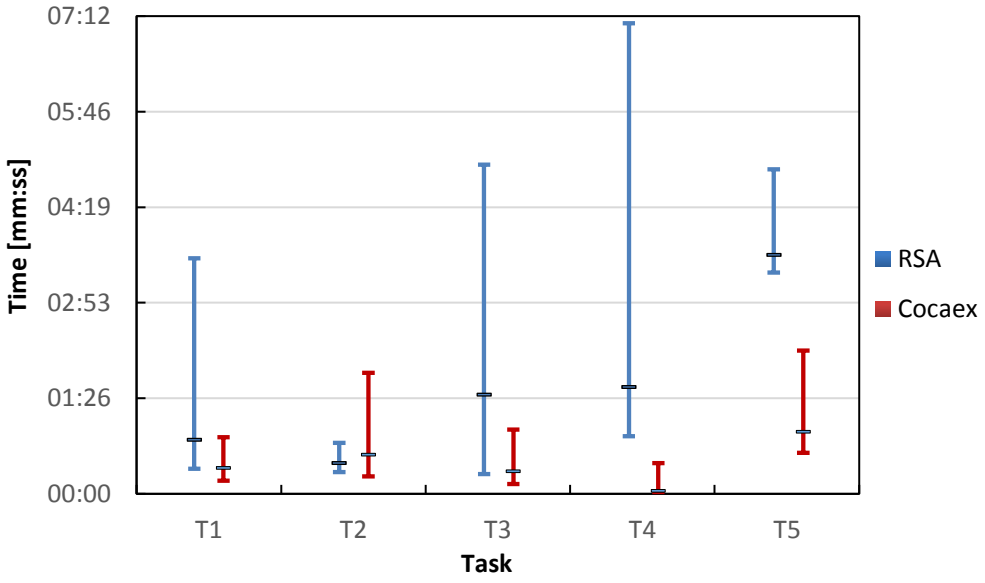


Figure 7.4: Minimum and Maximum Times with Marked Medians (black lines) Needed to Accomplish the Tasks in RSA and in CoCAEx

7.4 Lessons Learned

In this section we summarize observations regarding to component software visualization research based on current experience.

As stated in Section 6.2, we originally wanted to use a framework for visualization implementation. But closer look revealed problems connected with customization of third-party software. There was a time needed to understand a framework and even simple customizations were more difficult than without using it. On the other hand, features offered by frameworks were relatively quickly implemented from a scratch.

While implementing CoCAEx we verified that layout support is needed during the initial load of the diagram, but it is not beneficial for users to apply it after each change (e.g., component removal) in a diagram. It confuses a user and makes creation of mental model difficult.

While performing the user study we discovered that even users are well trained and able to use all necessary features of particular tool, they tend to use the features, which are easy to use (such as interactive highlighting), rather than complex ones.

Chapter 8

Conclusion

In this thesis, we suggested several criteria for evaluating tools targeted at visualization of component-based software. These criteria can be used on existing visualization tools as we presented on the example of IBM Rational Software Architect, which was evaluated with quite satisfactory results. On the other hand, this case shows that even advanced visualization tools currently address only a few of the needs related to component visualization.

The proposed criteria can thus also serve as a guideline for efforts towards better visualization of component-based applications. Currently the main problem behind the lack of such efforts can be due to relatively low usage of components. However, their importance continues to rise and future visualization tools should address these topics to a broader extent.

Advanced visualization techniques were described in this thesis. They help reduce the amount of lines in UML component diagram of large applications, by removing the selected components from the diagram area. It uses SeCo where the selected components are shown and symbolic delegates, which represent the connections instead of lines. A viewport technique was also described. This technique is used for showing all the information about interfaces for selected group of components right in the diagram area. The novel integration of above mentioned techniques was proposed. These techniques map a group of components to the content of a viewport. Viewport symbols for graphical representation of groups were also described. These symbols save a space in the diagram area. Appropriate interactions were proposed for all these techniques.

Above mentioned techniques are, among other benefits, useful in the reverse engineering process when the user is interactively getting familiar with the whole diagram. It helps with creating the mental model of the application by easing the process of clusters creation. This is the reason why these techniques use a ComAV platform, to perform reverse engineering of applications of various component models.

We also provided a proof the present component applications contain components with dense connections and thus the proposed technique would help reduce the clutter by moving them to the SeCo.

Most of the presented techniques are currently implemented in CoCAEx tool. The tool is web-based (HTML5, CSS3, JavaScript, jQuery and Java EE technologies were used) to be easy to use and provide even shorter reverse engineering process.

We performed a complete user study that compared performance in component diagram analysis tasks using two different component visualization approaches – experimental notation and UML. The experimental notation was used within the CoCAEx tool, which is implemented as a research proof of the concept, while UML is supported by a lot of commercial tools. Rational Software Architect was chosen to represent these tools because of its ability to easily study relations, which was most needed in this experiment.

In the user study, 12 participants were tested in a within-subject test with both tools. The participants accomplished 5 tasks with each tool. The main measure was the speed measured as the time needed to accomplish each task. We also collected subjective evaluation of the tool usage.

The data obtained shows that participants working with the CoCAEx tool are approximately 3× faster than those using UML in the RSA tool, though the RSA tool was faster in one task. The CoCAEx tool performed better in the remaining four tasks. We have discussed the reasons and showed the way of how different tasks could affect the overall performance.

Results of this user study therefore confirm that advanced visualization with a high level of interactivity is beneficial for users. What is more, the increased interaction required to uncover encapsulated information does not introduce significant slowdown.

The topic of thesis is still good research problem to continue a future work in this field and can be possibly generalized into other domains.

8.1 Evaluation of Thesis Goals

The main goal of the thesis was (as stated in Section 1.3):

> Bring better and more effective ways of large component software visualization to reduce the time needed to understand the application structure.

We introduced novel techniques design, provided their implemention and performed the evaluation, which shows that our approach reduced the time of reverse engineering process.

Two sub-goals and one approach defined in Section 1.3 were satisfied as stated below:

- sub-goal: **... not to modify the visual representation of UML component diagram ...** - The visual syntax was modified only for

added elements (groups, viewports) and in situations where interactivity is used (interface clustering, highlighting dependencies). Users recognized the syntax of components itself clearly, because it was not changed.

- sub-goal: **Visualize a sufficient amount of detail to be able to exactly identify individual components.** - Individual component contains its name in the diagram area as well as in SeCo. Only exceptions are unconnected components and collapsing a group in diagram area. In the first case components are considered as less important and thus hidden. In the latter case is its hiding selected intentionally (by collapsing a group) by user to hide unwanted details.

- approach: **We will provide ways to hide less important details and show them on demand.** - Interactivity is applied for interface clustering, groups collapsing, unconnected components, tracing dependencies, etc. As shown in Chapter 7, the interactivity speeds-up the tasks.

8.2 Future Work

The whole concept of used techniques is still evolving, which means that there are some techniques designed, but not yet implemented. Part of the future work will thus focus on implementation of such techniques, as described below.

We would like to improve the initial load presentation to be able to show a user simplified diagram for the first sight. For this purpose we would like to integrate clustering and automated removal of highly connected components. The former would allow forming the groups in an automated way and thus reducing the time now needed to form them. Also the optimal set of components, which should be removed to SeCo should be identified, based on the ideas presented in 8.2.1. The latter would reduce a need to select the most connected components manually based on the personal visual impression. It would also allow us to apply the layout on the graph, which visual clutter has been already reduced by the most connected components. Current state of this concept is described in 8.2.1.

Another visualization challenge is clustered interfaces exploration. There can be hundreds of interfaces hidden in one symbol. Showing them in simple list requires scrolling, which can confuse a user. Results of such research have large amount of applications in other domain, because a list element is widely used.

The viewport technique should be also fully implemented in the future, to allow us evaluate its impact.

There can be more components missing the same required interface. Visualization can help find unsatisfied dependencies directly in the diagram and reduce the clutter caused by their multiple occurrences by using advanced visualization techniques.

We believe that the presented ideas can be generalized to be used in other domains, where one suffers from visual clutter caused by the large number of

nodes and connection lines. Some of these are:

- bank transfers graphs,
- social networks graphs,
- server interconnections graphs.

Thus one part of the future work will be to provide examples of these applications including the technique adaptations.

8.2.1 Automated Removal of Highly Connected Components

It is clear that removing highly connected components helps to reduce a visual clutter in the diagram area (ignoring now the fact we lose some information). One of the research challenges in this area is how to find an optimal selection of components to be moved to SeCo in an automated way (for any given diagram). We would like to maximize the clutter reduction and minimize the number of components in SeCo.

In order to assess the degree of diagram clutter reduction, we need to define a function, which describes effectivity of each selection. Having such function would enable us to find its global optimum and thus determine or at least verify, whether we found the right selection. When constructing such function we should identify influencing factors, which could later be converted into its parameters.

There are many factors to be considered (some of them are have been proposed in [34]) when we want to formalize this problem. Some of them are even dependent on each other. The main factors related to our problem are:

1. angle of lines crossings,
2. transparency of particular element,
3. number of components,
4. number of lines,
5. lines length,
6. number of lines crossings,
7. number of lines crossing the nodes.

Calculating an effect of removing a particular selection of components we have to keep in mind, that it is valid only for current node layout. The node layout is also highly contributing factor for clutter reduction. For this section, we will consider a layout as given. Determining precise function for above mentioned factors is very complex problem. There are several options for solving such

problem. First we can analyse the effect of these factors separately and then analyse it again when combining them. This would require relatively large number of user studies. Second, we could establish a set of metrics for visual clutter that would allow us to perform the effect measurement in an automated way. Thus we would be able to determine the contribution of individual factors to the overall clutter, as well as desired (non-trivial) function. The problem is that such metrics, which are reliable and solid, are not widely established. Their identification would also be an interesting research challenge suitable for future work.

By increasing first two factors, the clutter is lowered. On the other for factors 3.-7. the clutter is reduced by their decreasing. For instance, decreasing the number of components in the diagram area and thus moving them to SeCo obviously decreases the visual clutter in the diagram area.

Increasing the number of components in SeCo does not increase the visual clutter much for first few components. But when the number crosses the limit of visible components in SeCo and a user is forced to use a scrollbar, then the usability is lowered. We can consider this usability complication as similar to one a user has when having a diagram cluttered. Considering these effects caused by number of removed components from SeCo to overall usability of the whole application, we could establish a function. Implementing such function could provide us a possibility to find suitable solution for the clutter reduction.

Bibliography

[1] Metricview, 2011. http://swerl.tudelft.nl/twiki/pub/Main/BENEVOL2006program/ChristianLange.pdf, Accessed: 2014-09-29.

[2] Handbook of graph drawing and visualization, 2012. http://www.cs.brown.edu/~rt/gdhandbook/, Accessed: 2014-09-29.

[3] Java universal network/graph framework, 2012. http://jung.sourceforge.net/, Accessed: 2014-09-29.

[4] The stony brook algorithm repository, 2012. http://www.cs.sunysb.edu/~algorith/implement/Java.shtml, Accessed: 2014-09-29.

[5] Enterprise architect, 2014. Accessed: 2014-09-29.

[6] Klay layered layout, 2014. http://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/KLay+Layered, Accessed: 2014-09-29.

[7] Magicdraw, 2014. Accessed: 2014-09-29.

[8] CoCoME implementation in OSGi, 2014. Accessed: 2014-09-29.

[9] IBM Rational Software Architect, 2014. Accessed: 2014-09-29.

[10] Visual paragigm, 2014. Accessed: 2014-09-29.

[11] yworks layouts gallery, 2014. http://www.yworks.com/en/products_yfiles_practicalinfo_gallery.html, Accessed: 2014-09-29.

[12] Jan Øyvind Aagedal. *Quality of Service Support in Development of Distributed Systems*. PhD thesis, University of Oslo, 2001.

[13] Sazzadul Alam and Philippe Dugerdil. EvoSpaces Visualization Tool: Exploring Software Architecture in 3D. In *WCRE '07: Proceedings of the 14th Working Conference on Reverse Engineering*, pages 269–270, Washington, DC, USA, 2007. IEEE Computer Society.

[14] Felix Bachmann, Len Bass, Charles Buhman, Santiago C. Dorda, Fred Long, John Robert, Robert Seacord, and Kurt Wallnau. Volume ii: Technical concepts of component-based software engineering, 2nd edition. Technical report, CMU/SEI - Carnegie Mellon University/Software Engineering Institute, 2000.

[15] 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.

[16] 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.

[17] Benjamin B. Bederson and Angela Boltman. Does animation help users build mental maps of spatial information? In *Proceedings of the 1999 IEEE Symposium on Information Visualization*, INFOVIS '99, pages 28–, Washington, DC, USA, 1999. IEEE Computer Society.

[18] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. *Computer*, 32(7):38–45, 1999.

[19] Antoine Beugnard, Jean-Marc Jézéquel, and Noël Plouzeau. Contract aware components, 10 years after. In Javier Cámara, Carlos Canal, and Gwen Salaün, editors, *WCSI*, volume 37 of *EPTCS*, pages 1–11, 2010.

[20] Roberto Almeida Bittencourt and Dalton Dario Serey Guerrero. Comparison of graph clustering algorithms for recovering software architecture module views. In *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*, CSMR '09, pages 251–254, Washington, DC, USA, 2009. IEEE Computer Society.

[21] 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.

[22] Franz J. Brandenburg, Michael Himsolt, and Christoph Rohrer. An experimental comparison of force-directed and randomized graph drawing algorithms. pages 76–87. Springer-Verlag, 1996.

[23] 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.

[24] 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.

[25] 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.

[26] Pierre Caserta and Olivier Zendra. Visualization of the static aspects of software: A survey. *IEEE Trans. Vis. Comput. Graph.*, 17(7):913–933, 2011.

[27] K. Cassell, C. Anslow, L. Groves, P. Andreae, and S. Marshall. Visualizing the refactoring of classes via clustering. In Mark Reynolds, editor, *Australasian Computer Science Conference (ACSC 2011)*, volume 113 of *CRPIT*, pages 63–72, Perth, Australia, 2011. ACS.

[28] Chaomei Chen. Graph drawing algorithms. In *Information Visualization*, pages 65–87. Springer London, 2006. 10.1007/1-84628-579-8_3.

[29] Yves Chiricota, Fabien Jourdan, and Guy Melançon. Software components capture using graph clustering. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, IWPC '03, pages 217–. IEEE Computer Society, 2003.

[30] Andy Cockburn, Amy Karlson, and Benjamin B. Bederson. A review of overview+detail, zooming, and focus+context interfaces. *ACM Comput. Surv.*, 41(1):2:1–2:31, January 2009.

[31] 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.

[32] Ivica Crnkovic;, Severine Sentilles, Aneta Vulgarakis, and Michel R.V. Chaudron. A classification framework for software component models. *IEEE Transactions on Software Engineering*, 37:593–615, 2011.

[33] Tim Dwyer, Bongshin Lee, Danyel Fisher, Kori Inkpen Quinn, Petra Isenberg, George Robertson, and Chris North. A comparison of user-generated and automatic graph layouts. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):961–968, November 2009.

[34] G. Ellis and A. Dix. A taxonomy of clutter reduction for information visualisation. *Visualization and Computer Graphics, IEEE Transactions on*, 13(6):1216 –1223, nov.-dec. 2007.

[35] Jean-Marie Favre and Humberto Cervantes. Visualization of component-based software. In *Proceedings of the 1st International Workshop on Visualizing Software for Understanding and Analysis*, pages 51–, Washington, DC, USA, 2002. IEEE Computer Society.

[36] X. Franch. Systematic formulation of non-functional characteristics of software. In *Proceedings of International Conference on Requirements Engineering (ICRE)*, pages 174–181, 1998.

[37] Mathias Frisch and Raimund Dachselt. Off-screen visualization techniques for class diagrams. In *Proceedings of the 5th international symposium on Software visualization*, SOFTVIS '10, pages 163–172, New York, NY, USA, 2010. ACM.

[38] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Softw. Pract. Exper.*, 21(11):1129–1164, November 1991.

[39] Emden Gansner, Yifan Hu, Stephen Kobourov, and Chris Volinsky. Putting recommendations on the map: visualizing clusters and relations. In *Proceedings of the third ACM conference on Recommender systems*, RecSys '09, pages 345–348, New York, NY, USA, 2009. ACM.

[40] Emden R. Gansner, Yifan Hu, Stephen C. North, and Carlos Eduardo Scheidegger. Multilevel agglomerative edge bundling for visualizing large graphs. In Giuseppe Di Battista, Jean-Daniel Fekete, and Huamin Qu, editors, *PacificVis*, pages 187–194. IEEE, 2011.

[41] Martin Glinz. On non-functional requirements. In *Requirements Engineering Conference*, pages 21–26, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[42] Xiaohui Gu, Klara Nahrstedt, Wanghong Yuan, Duangdao Wichadakul, and Dongyan Xu. An xml-based quality of service enabling language for the web. *Journal of Visual Language and Computing, Special Issue on Multimedia Language for the Web*, 13:61–95, 2001.

[43] Stefan Hachul and Michael Jünger. Large-graph layout algorithms at work: An experimental study. *Journal of Graph Algorithms and Applications*, 11(21):345–369, 2007.

[44] 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.

[45] David Harel and Yehuda Koren. A fast multi-scale method for drawing large graphs (full version). In *Journal of Graph Algorithms and Applications*, pages 183–196. Springer-Verlag, 2000.

[46] Jeffrey Heer, Michael Bostock, and Vadim Ogievetsky. A tour through the visualization zoo. *Commun. ACM*, 53(6):59–67, June 2010.

[47] Michael Himsolt. Comparing and evaluating layout algorithms within graphed. *J. Visual Languages and Computing*, 6:255–273, 1995.

[48] Ric Holt. Software architecture as a shared mental model. In *Proceedings of International Workshop on Program Comprehension*, 2002.

[49] Danny Holten. *Visualization of Graphs and Trees for Software Analysis.* PhD thesis, Technische Universiteit Eindhoven, 2009.

[50] Danny Holten, Bas Cornelissen, and Jarke J. van Wijk. Trace visualization using hierarchical edge bundles and massive sequence views. *Visualizing Software for Understanding and Analysis, International Workshop on*, 0:47–54, 2007.

[51] Danny Holten and Jarke J. van Wijk. Force-directed edge bundling for graph visualization. *Comput. Graph. Forum*, 28(3):983–990, 2009.

[52] Danny Holten and Jarke J. van Wijk. A user study on visualizing directed edges in graphs. In *Proceedings of the 27th international conference on Human factors in computing systems*, CHI 09, pages 2299–2308, New York, NY, USA, 2009. ACM.

[53] L. Holy, J. Snajberk, and P. Brada. Lowering visual clutter of clusters in component diagrams. In *Proceedings of International Conference on Software Engineering Advances*, pages 304–307, Red Hook, NY, USA, November 2012. IARIA.

[54] L. Holy, J. Snajberk, and P. Brada. Visual clutter reduction for uml component diagrams: A tool presentation. In *Visual Languages and Human-Centric Computing (VL/HCC), 2012 IEEE Symposium on*, pages 253–254, Sept 2012.

[55] L. Holy, J. Snajberk, P. Brada, and K. Jezek. A visualization tool for reverse engineering of complex component applications. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 500–503, Sept 2013.

[56] Lukas Holy and Premek Brada. Viewport for component diagrams. In Marc J. van Kreveld and Bettina Speckmann, editors, *Graph Drawing*, volume 7034 of *Lecture Notes in Computer Science*, pages 443–444. Springer, 2011.

[57] Lukas Holy, Kamil Jezek, Jaroslav Snajberk, and Premek Brada. Lowering visual clutter in large component diagrams. In Ebad Banissi, Stefan Bertschi, Camilla Forsell, Jimmy Johansson, Sarah Kenderdine, Francis T. Marchese, Muhammad Sarfraz, Liz J. Stuart, Anna Ursyn, Theodor G. Wyeld, Hanane Azzag, Mustapha Lebbah, and Gilles Venturini, editors, *IV*, pages 36–41. IEEE Computer Society, 2012.

[58] Lukas Holy, Jaroslav Snajberk, and Premek Brada. Evaluating component architecture visualization tools - criteria and case study. In *GRAPP/IVAPP*, pages 737–742, 2012.

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

[60] International Standard Organization (ISO/IEC). Informational technology – product quality – part 1: Quality model. International Standard ISO/IEC 9126, June 2001.

[61] T. J. Jankun-Kelly and Kwan-Liu Ma. Moiregraphs: radial focus+context visualization and interaction for graphs with visual nodes. In *Information Visualization, 2003. INFOVIS 2003. IEEE Symposium on*, pages 59–66, Washington, DC, USA, Oct 2003. IEEE Computer Society.

[62] Kamil Ježek and Premek Brada. Correct matching of components with extra-functional properties - a framework applicable to a variety of component models. In *Evaluation of Novel Approaches to Software Engineering (ENASE)*. SciTePress, 2011. ISBN: 978-989-8425-65-2.

[63] K. Jezek, L. Holy, and P. Brada. Static component compatibility visualisation for various component models. In *Visual Languages and Human-Centric Computing (VL/HCC), 2013 IEEE Symposium on*, pages 191–192, Sept 2013.

[64] Kamil Jezek, Premek Brada, and Lukas Holy. Enhancing OSGi with Explicit, Vendor Independent Extra-Functional Properties. In *TOOLS (50)*, volume 7304 of *Lecture Notes in Computer Science*, pages 108–123. Springer, 2012.

[65] 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.

[66] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Inf. Process. Lett.*, 31(1):7–15, April 1989.

[67] Christian Klein and Benjamin B. Bederson. Benefits of animated scrolling. In *CHI '05 extended abstracts on Human factors in computing systems*, CHI EA '05, pages 1965–1968, New York, NY, USA, 2005. ACM.

[68] Ralf Kollman, Petri Selonen, Eleni Stroulia, Tarja Systä, and Albert Zündorf. A study on the current state of the art in tool-supported uml-based static reverse engineering. In Arie van Deursen and Elizabeth Burd, editors, *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE 2002)*. IEEE Computer Society, November 2002.

[69] Adrian Kuhn, David Erni, Peter Loretan, and Oscar Nierstrasz. Software cartography: thematic software visualization with consistent layout. *J. Softw. Maint. Evol.*, 22:191–210, April 2010.

[70] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[71] C. F.J Lange, M. R.V Chaudron, and J. Muskens. In practice: UML software architecture and design description. *IEEE Software*, 23(2):40– 46, April 2006.

[72] Kung-Kiu Lau and Vladyslav Ukis. Defining and checking deployment contracts for software components. *In Proceedings of the 9th International Symposium on Component-Based Software Engineering, volume 4063 of LNCS*, pages 1–16, 2006.

[73] Jiri Loudil. Extra-functional properties visualization, 2014.

[74] I.S. MacKenzie. *Human-Computer Interaction: An Empirical Research Perspective*. Elsevier Science, 2012.

[75] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Proceedings of the 6th International Workshop on Program Comprehension*, IWPC '98, pages 45–, Washington, DC, USA, 1998. IEEE Computer Society.

[76] Fintan McGee and John Dingliana. Visualising small world graphs - agglomerative clustering of small world graphs around nodes of interest. In Paul Richard, Martin Kraus, Robert S. Laramee, and José Braz, editors, *GRAPP/IVAPP*, pages 678–689. SciTePress, 2012.

[77] Nenad Medvidovic, David S. Rosenblum, David F. Redmiles, and Jason E. Robbins. Modeling software architectures in the unified modeling language. *ACM Trans. Softw. Eng. Methodol.*, 11(1):2–57, January 2002.

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

[79] Mubarak Mohammad and Vasu S. Alagar. TADL - an architecture description language for trustworthy component-based systems. In *ECSA '08: Proceedings of the 2nd European conference on Software Architecture*, pages 290–297. Springer, 2008.

[80] Daniel Moody and Jos van Hillegersberg. Evaluating the visual syntax of uml: An analysis of the cognitive effectiveness of the uml family of diagrams. In Dragan Gasevic, Ralf Lammel, and Eric Van Wyk, editors, *Software Language Engineering*, volume 5452 of *Lecture Notes in Computer Science*, pages 16–34. Springer Berlin Heidelberg, 2009.

[81] Daniel Moody and Jos van Hillegersberg. Evaluating the visual syntax of UML: An analysis of the cognitive effectiveness of the UML family of diagrams. In Dragan Gasevic, Ralf Lammel, and Eric Van Wyk, editors, *Software Language Engineering*, volume 5452 of *Lecture Notes in Computer Science*, pages 16–34. Springer Berlin / Heidelberg, 2009.

[82] S. Morris and G. Spanoudakis. Uml: an evaluation of the visual syntax of the language. In *System Sciences, 2001. Proceedings of the 34th Annual Hawaii International Conference on*, pages 10 pp.–, Washington, DC, USA, Jan 2001. IEEE Computer Society.

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

[84] OMG. CORBA components. OMG Specification formal/02-12-06, Object management Group 2006, 2006.

[85] OMG. UML 2.4 specification. OMG document ptc/2010-11-14, Object Management Group 2011, 03 2011.

[86] OSGi Alliance. OSGi service platform v4.2. Core specification, OSGi Alliance 2009, 2009.

[87] Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *IEEE Trans. Software Eng*, 28(11):1056–1076, 2002.

[88] Helen C. Purchase, Matthew McGill, Linda Colpoys, and David Carrington. Graph drawing aesthetics and the comprehension of uml class diagrams: an empirical study. In *Proceedings of the 2001 Asia-Pacific symposium on Information visualisation - Volume 9*, APVis '01, pages 129–137, Darlinghurst, Australia, Australia, 2001. Australian Computer Society, Inc.

[89] D. Rafiei. Effectively visualizing large networks through sampling. In *Visualization, 2005. VIS 05. IEEE*, pages 375 – 382, oct. 2005.

[90] Ratneshwer and A. K. Tripathi. Dependence analysis of software component. *SIGSOFT Softw. Eng. Notes*, 35:1–9, July 2010.

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

[92] Ruth Rosenholtz, Yuanzhen Li, and Lisa Nakano. Measuring visual clutter. *Journal of Vision*, 7(2), August 2007.

[93] S.E. Schaeffer. Graph clustering. *Computer Science Review*, 1(1):27–64, 2007.

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

[95] Séverine Sentilles, Petr Stepan, Jan Carlson, and Ivica Crnkovic. Integration of extra-functional properties in component models. *12th International Symposium on Component Based Software Engineering (CBSE 2009), LNCS 5582*, June 2009.

[96] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Proceedings of the 1996 IEEE Symposium on Visual Languages*, VL '96, pages 336–, Washington, DC, USA, 1996. IEEE Computer Society.

[97] Keng Siau and Yuhong Tian. A semiotic analysis of unified modeling language graphical notations. *Requirements Engineering*, 14:15–26, 2009. 10.1007/s00766-008-0071-7.

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

[99] Jaroslav Snajberk, Lukas Holy, and Premek Brada. Comav - a component application visualisation tool. In *Proceedings of International Conference on Information Visualization Theory and Applications*. SciTePress, 2012.

[100] M. A D Storey, K. Wong, and H.A Muller. How do program understanding tools affect how programmers understand programs? In *Reverse Engineering, 1997. Proceedings of the Fourth Working Conference on*, pages 12–21, Oct 1997.

[101] Sun Microsystems. Enterprise JavaBeans(TM), version 3.0. EJB Core, Sun Microsystems, 2006, 2006.

[102] James Surowiecki. *The Wisdom of Crowds*. Anchor, 2005.

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

[104] 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.

[105] Alexandru Telea, Lucian Voinea, and Hans Sassenburg. Visual tools for software architecture understanding: A stakeholder perspective. *IEEE Softw.*, 27:46–53, November 2010.

[106] Scott Tilley. Documenting software systems with views vi: Lessons learned from 15 years of research & practice. In *Proceedings of the 27th ACM International Conference on Design of Communication*, SIGDOC '09, pages 239–244, New York, NY, USA, 2009. ACM.

[107] Barbara Tversky, Julie Bauer Morrison, and Mireille Betrancourt. Animation: can it facilitate? *Int. J. Hum.-Comput. Stud.*, 57(4):247–262, October 2002.

[108] J. VANWIJK and W NUIJ. Amodel for smooth viewing and navigation of large 2d information spaces. In *IEEE Trans. Visual. Comput. Graph. 10, 4,*, page 447–458. IEEE, 2004.

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

[110] Chris Walshaw. A multilevel algorithm for force-directed graph drawing. In *Proceedings of the 8th International Symposium on Graph Drawing*, GD '00, pages 171–182, London, UK, UK, 2001. Springer-Verlag.

[111] Jingwei Wu, Ahmed E. Hassan, and Richard C. Holt. Comparison of clustering algorithms in the context of software evolution. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ICSM '05, pages 525–535, Washington, DC, USA, 2005. IEEE Computer Society.

Appendix A

Deployment and Availability

The CoCAEx application is available online[1] for testing. For the demonstration purposes a demo diagram or a public diagrams can be used, as shown in Figure A.1.

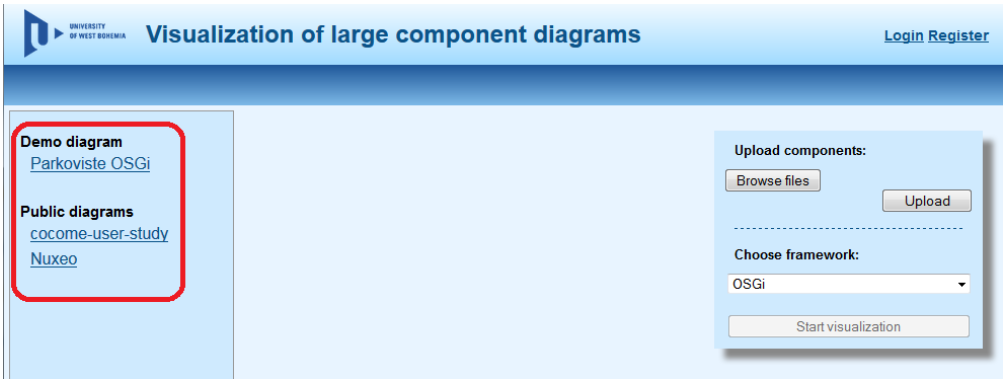


Figure A.1: Demo and Public Diagrams

There is also a possibility to use the corpus of test data files[2] to be uploaded to the CoCAEx server. The corpus needs to be extracted and components (.jar files) uploaded.

In the first step, we select the extracted bundles and upload them to the server via upload dialog. To do that we select *Browse...*, then we select all bundles and click *Open* button (or adequate one for different languages). Files are selected and we press *Upload* button. Files are uploaded to the server (which can take a while). After that, we start visualization by clicking the *Start visualization* button and the reverse-engineered diagram will be shown.

All necessary information and screencasts are described on the information webpage[3].

[1] `http://relisa-dev.kiv.zcu.cz:8083/efpcocaex/`
[2] `http://relisa-dev.kiv.zcu.cz/data/cocaex/demo_data.zip`
[3] `http://relisa.kiv.zcu.cz/areas/large-diagrams-visualization.html`

Appendix B

List of Published Articles

The following papers were submitted to be published in journals:

1. Holý, L., Malý, I., Čmolík, L., Ježek, K., and Brada P.: **An Interactive UML-like Visualization for Large Software Diagrams**, Research Journal of Applied Sciences, Engineering and Technology, Elsevier (Scopus), 11(4) October 2015

The following papers are essential for my work.

1. Holý, L. and Brada P.: **Viewport for component diagrams**, Proceedings of the 19th International Conference on Graph Drawing, Eindhoven, The Netherlands, Springer, 2011

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

3. Holý, L., Ježek, K., Šnajberk, J. and Brada P.: **Lowering Visual Clutter in Large Component Diagrams**, Proceedings of International Conference on Information Visualization, Montpellier, France, IEEE Computer Society, 2012

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

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

6. Holý, L., Šnajberk, J., Brada P. and Ježek, K.: **A Visualization Tool for Reverse-engineering of Complex Component**

Applications, Proceedings of the 29th IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, IEEE Computer Society, 2013

The following papers were published in conference proceedings, paper **An Advanced Interactive Visualization Approach for Component-Based Software: A User Study** recieved a Best Paper Award:

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

2. Šnajberk, J., Holý, L., Ježek, K. and Brada P.: **An Advanced Interactive Visualization Approach for Component-Based Software: A User Study**, Proceedings of International Conference on Software Engineering Advances, Lisbon, Portugal, IARIA, 2012

3. Šnajberk, J., Holý, L. and Brada P.: **Visualization of Component-Based Applications Structure using AIVA**, Proceedings of European Conference on Software Maintenance and Reengineering, Genova, Italy, IEEE Computer Society, 2013

4. Ježek, K., Brada P. and Holý, L.: **Enhancing OSGi with Explicit, Vendor Independent Extra-functional Properties**, Proceedings of the 50th International Conference on Objects, Models, Components, Patterns, Prague, Czech Republic, Springer, 2012

5. Ježek, K., Holý, L. and Brada P. : **Dependency Injection Refined by Extrafunctional Properties**, Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing, Innsbruck, Austria, IEEE Computer Society, 2012

6. Ježek, K., Holý, L. and Brada P. : **Supplying Compiler's Static Compatibility Checks by the Analysis of Third-party Libraries**, Proceedings of European Conference on Software Maintenance and Reengineering, Genova, Italy, IEEE Computer Society, 2013

7. Ježek, K., Holý, L. and Brada P. : **Static component compatibility visualisation for various component models**, Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing, San Jose, USA, IEEE Computer Society, 2013

8. Ježek, K., Holý, L., Slezáček, A., and Brada P. : **Software Components Compatibility Verification Based on Static Byte-Code Analysis**, Proceedings of the 39th EUROMICRO Conference on Software Engineering and Advanced Applications, Santander, Spain, IEEE Computer Society, 2013