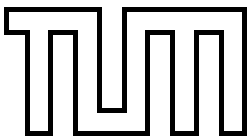
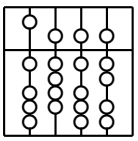


Technische Universität München
Fakultät für Informatik



Systementwicklungsprojekt

Run-time Development and Configuration of Dynamic Service Networks

Markus Michael Geipel

Aufgabensteller: Prof. Bernd Brügge Ph.D. , Prof. Gudrun Klinker Ph. D.

Betreuer: Dipl.-Inf. Asa MacWilliams, Dipl.-Inf. Christian Sandor

Abgabedatum: Juni 2004

Erklärung

Ich versichere, dass ich diese Ausarbeitung des Systementwicklungsprojekts selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 29. Juni 2004 Markus Michael Geipel

Abstract

Development and configuration of dynamic service networks are core activities in the development process of new DWARF applications. This development and configuration can be tedious if done only with text editors and the console. The following work describes how the monitoring tool DIVE has been restructured and extended, based on experience gained during the CAR project, to allow "Run-time Development and Configuration of Dynamic Service Networks". Furthermore the results will be discussed as well as possible future work.

Contents

Chapter 1

Introduction

"The nets are vast and infinite."
Masamune Shirow and Mamoru Oshii, *Ghost in the Shell*

1.1 Augmented Reality and Service Networks

One of the main issues of an augmented reality system is the processing of streams of data. Cameras deliver video data, these have to be processed via image recognition to position data, which is needed by the renderer. The renderer itself, aggregating all necessary data, produces video data which is feed back to the user. To deal with these streams of data, a system of dynamically cooperating services is an obious solution **??**. The question now is: How can such a network be built and configured? In order to formulate the concrete problem statement, we will first consider the specific context of this project ...

1.2 Context

1.2.1 The DWARF Framework

The DWARF Framework is the basis of this work. It consists of dynamically cooperating services. A node in this network is called Service and forms the basic processing unit. An edge is a data stream. Every Service may have Needs and Abilities. Needs are data sinks and Abilities are data sources. Every data stream has a type: for example "PoseData". The whole Service is described via a ServiceDescription. It holds the service name, information about Needs and Abilities as well as annotating information in form of Attributes. These Attributes can be used to control the connection behavior based on meta information. CORBA based middleware manages the run-time connection of services. A so-called Service Manager runs on each host in the DWARF network. It holds all Service Descriptions of local Services and tries to connect them in coordination with the other Service Managers on other hosts. The service network is thus: highly distributed, transparent and

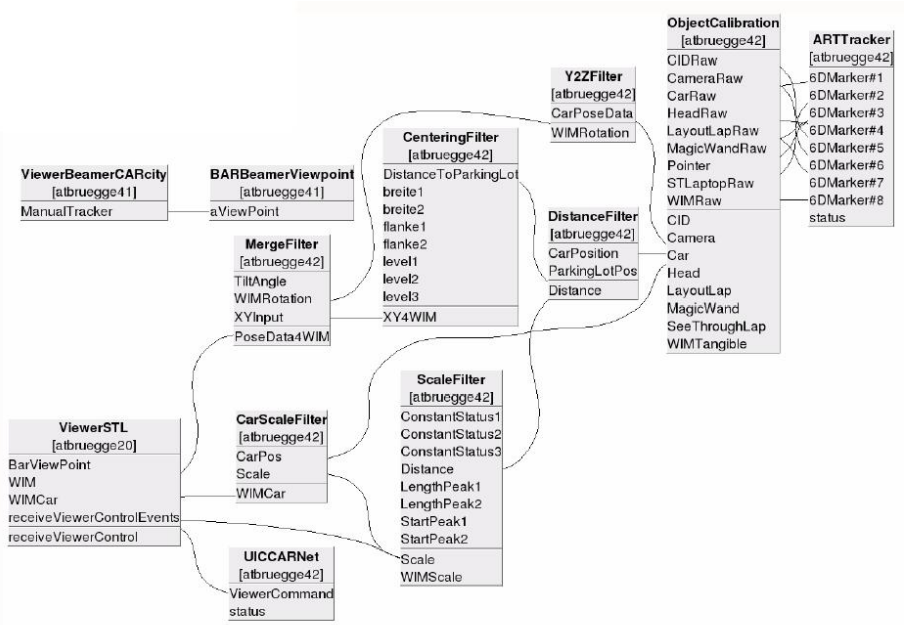


Figure 1.1: A network of cooperating services established in the CAR-Project with the DWARF-System and visualized via DIVE.

heterogeneous concerning operating system as well as programming language. It is essential to be familiar with this concept in order to follow the further discussion. Additional information can be found in [1, 12]. [13] gives an example of application development in DWARF.

1.2.2 The Monitoring Tool DIVE

As fascinating as the architecture of DWARF is, at the first glimpse, it seems to comprise a problem for the developer. Daniel Pustka the developer of DIVE puts it this way:

> Finding out what services are running on the various computers and how the DWARF middleware has connected them currently is a tedious task. For that purpose, a developer usually would open terminal windows that contain the diagnostic output of all participating software components (the service manager on the need's side and the two services). As the output most of the time also contains other data, finding the right information there is difficult. [16]

In order to use the full power of DWARF, this problem had to be solved. So this problem statement became the starting point of DIVE, the DWARF Interactive Visualization Environment, a kind of monitoring tool for DWARF. Via DIVE, the service network becomes visible for the user, be it the developer or just an interested spectator: An automatically laid out graph shows the network including Needs, Abilities and their Attributes. Further

more data streams can be bugged. But still the 'interactive' 'I' in DIVE is not supported with functionality, although is was indended this way. This interaction, or better to say, missing interaction leads us back to the initial question: How can such a network be built and configured? And straight to a follow-up question: How do we know what interaction is needed. The answer to the second question is be given by the CAR-Project ...

1.2.3 The CAR Project

The CAR Project is an AR Project based on the DWARF System. The interesting points are: First, CAR is concerned with authoring for AR, so the need to integrate an extended version of DIVE is quite obvious. Second, CAR involves approximately ten persons of different skill level concerning DWARF and AR, so it is the ideal test and inspiration environment for extending DIVE. Figure 1.2 shows a part of the CAR-Setup: The image in the upper right corner shows a tangible toy car that can be moved on a projected city map. The big picture shows the simulated view through the windshield of this car.

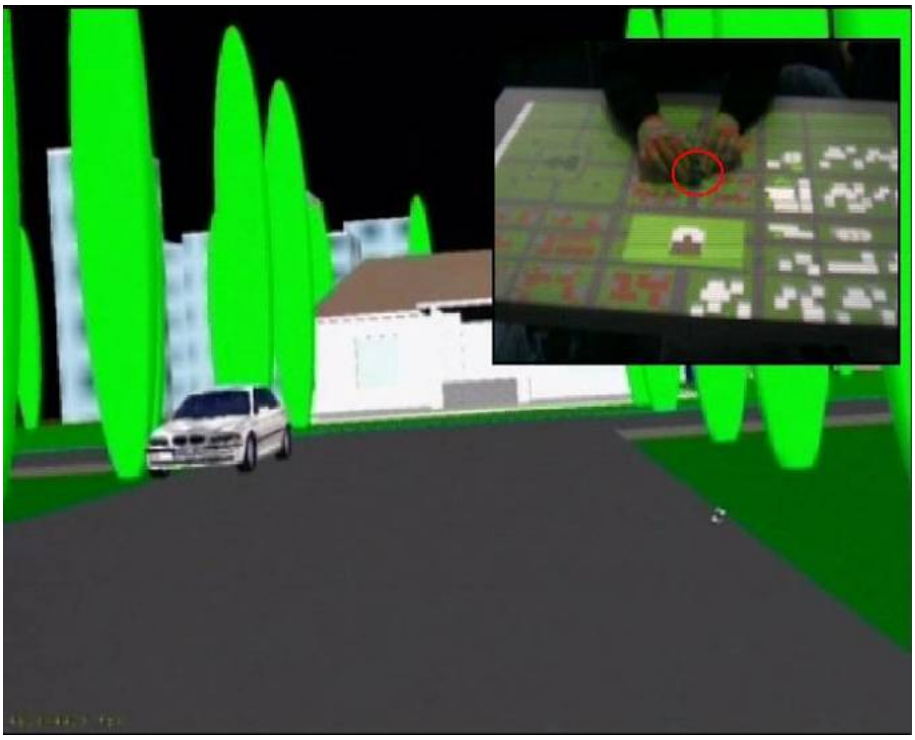


Figure 1.2: The CAR-Setup.

So, what is CAR exactly about? Here is the official problem statement:

> The goal of CAR is to create a collaboration platform for computer scientists (UI programmers) and non-technicians (human factors, psychologists etc.). The platform allows collaborative design of visualizations and interaction metaphors to be used in the next-generation cars with Head-Up Displays. We focus on

> two scenarios: parking assistance and a tourist guide. On the technical level we try to incorporate techniques like: layout of information on multiple displays, active user interfaces based on user modelling with eye tracking and an improved User Interface Controller with a rapid prototyping GUI. Additionally a dynamically configurable set of filters (each having an appropriate GUI for tuning parameters) is provided [...] [22]

1.3 Problem Statement

Before the beginning of my work, DIVE was a valuable monitoring tool for DWARF, but only this, a monitoring tool. Service networks have to be configured and developed, not only monitored. The process of configuring and developing service networks is tedious and time consuming, like monitoring was before DIVE. Shells and Editors have to be used and services or whole parts of the service network have to be restarted which makes the feedback circle, "try-change-try", very sluggish and discouraging. So it is self-evident that the 'I' in DIVE has to be filled with meaning: Main properties of the service network have to be changeable via DIVE during run-time. To accomplish this, information of the service network has to be accessible in a more convenient way: faster and ergonomically. The developement of DIVE by Daniel Pustka was done in the context of a "System Entwicklungs Projekt" (SEP). Also my work, the extention and further developement of DIVE, will be done in the context of a SEP.

Chapter 2

Requirements Analysis

> Any sufficiently advanced bug is indistinguishable from a feature.
> *Rich Kulawiec*

2.1 Requirements

As this SEP is not a greenfield engineering project but a re-engineering project, most requests are feature requests. But we will see that not only extensions are needed but also architectural changes: for example a faster update procedure. The requirements for this SEP came mainly from three sources:

1. Meetings of CAR-Probect members
2. Feature requests published by DWARF users in the Wiki-Web
3. Meetings with my advisors

As it was mentioned, the CAR Project was the main source of new feature ideas. All the Features that are listed in the following section were of course considered necessary and valuable in the beginning of the project. The success of each individual feature is analyzed and discussed in Chapter 5.

2.1.1 Functional Requirements

Changing Attributes and Predicates Attributes and Predicates determine the connection establishment made by the middleware. Furthermore attributes provide meta-information that can be used by the application as well as by the developer. Thus enabling DIVE to manipulate Attributes and Predicates is the main step towards interactive configuration and development of service networks.

Connecting Services This may at first sound misleading: only the middleware connects services and this only on the basis of the requested and provided data, the communication protocol and the matching of Attributes and Predicates. So connecting

Services in reality only means to adjust the Attributes and Predicates in a way that the requested connection may be established by the middleware. It is a kind of automatism that saves the user time.

Starting Services The Developer should be able to access information about what Services may be started and be enabled to start them on click.

Configuring Services A configuration beyond the change of Attributes and Predicates is meant here. Some Services cannot be configured by just changing same Attributes that can only host strings. Maybe a complete GUI is needed. There has to exist a flexible extension mechanism to open custom configuration tools from within DIVE. The first idea of a configuration of Services within DIVE was introduced by Daniel Pustka [16].

Making Changes Persistent Manipulating, configuring and developing a service network would be in vein, if no mechanism existed to make the changes persistent. So there has to be a way to save the altered ServiceDescriptions for later use.

List View Also this idea was first found in Daniel Pustka's original work [16]. The idea is to provide an alternative view on the system, that does not suffer the complexity of network graphs, a view in which Services can easily be found by name or even sorted by name or same other criteria like a specific Attribute. Of course in the List View the connection structure gets lost. So a List View can only be an addition to the Graph View.

Grouped View In order to grasp the big picture, it makes sense to reduce the complexity. This can be done by grouping Services based on one of their Attributes. For example grouping them by hostname to see which host is heavily loaded or which hosts communicate with each other.

UML like Graph View The Graph View should be compliant with the UML-Standard [5]. Please note, that this requirement has not been implemented yet. Further information can be found in section 6.

2.1.2 Nonfunctional Requirements

Performance The constraints concerning update speed in an interactive tool are much stricter than for a monitoring tool. For effective work even a large DWARF network should not lead to delays of more than approximately five seconds in average. As a reference DWARF-Application we choose the CAR-Application. ??? Nr. services in the CAR?

Usability The effectiveness of DIVE is not only determined by the functions it provides but also who quick they can be accessed: Often used functions, like update, should be accessible via one click on the GUI. The UI and visualization elements should be supported in their expressiveness by icons and color schemes.

Reliability In order to work productively with DIVE, a certain stability should be guaranteed. It is hard to exactly define this "certain" because there are no strict reliability levels to separate acceptable stability from unacceptable stability. As a rule of thumb, DIVE should run for several hours in productive work, without restart [1].

Supportability To make support and further changes and extensions easier, all new features should by commented using doxigen [23].

2.1.3 Pseudo Requirements

Implementation As the new features are integrated in the already existing DIVE, the same implementation constraints have to be considered. Mainly the compatibility with the following software, according to [16]: Linux 2.4, GNU autotools (automake, autoconf), OmniORB. New constraints are of course the usage of the same programming language that was used for the already existing DIVE and the same windowing toolkit: C++ [19], in combination with STL [18] and the windowing toolkit QT [21]. Last but not least, DIVE is part of DWARF and thus has to be incorporated in the DWARF infrastructure: it has to be available in the CVS-repository of DWARF and online documentation and discussion of DIVE has to take place on DWARF's Wiki-Web [4].

Interface The extensions should use the extension mechanism for DIVE that was introduced by Daniel Pustka. Furthermore the interaction with the DWARF network has used the interfaces declared by DWARF [4] and CORBA [8].

Packaging As the DWARF System is open source software, published under the General Public Licence [6]. This is also true for DIVE. The distribution of DWARF is done via a public CVS-Repository. Also DIVE has been published this way and will still be published this way.

2.2 Use Case Models

How will the Features be constituted in the functional requirements list work for the user? This will shown by the following scenarios and use cases. Figure 2.1 gives an overview of the use cases.

[1] Taking into account the stability of certain past software products of a certain big software company, this is already a commonly accepted standard.

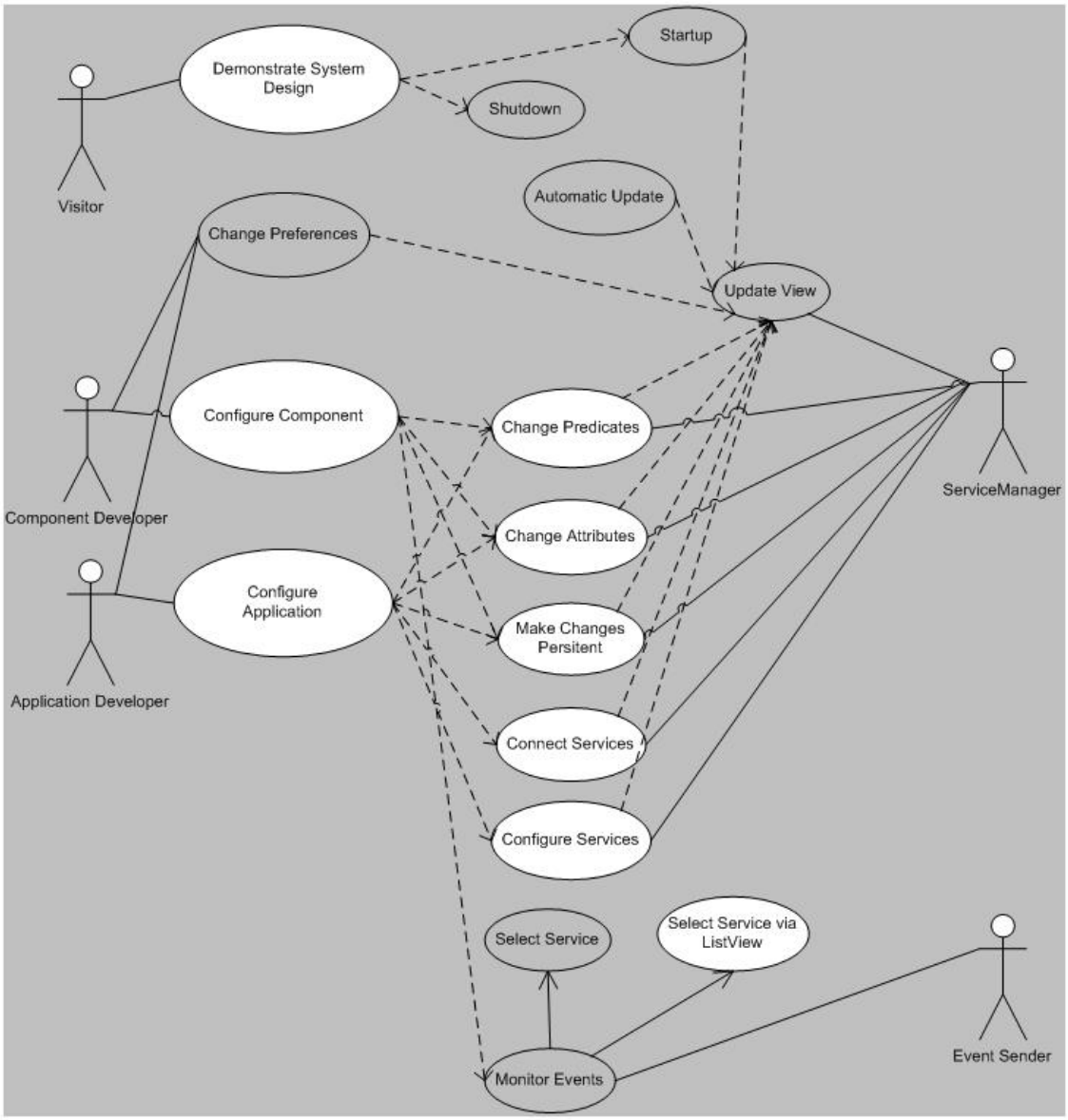


Figure 2.1: Use case overview. Use cases that where added or changed are marked white.

2.2.1 Scenarios

Configuring the whole Service Network

The developer Maxi[2] wants to write an application that shows a world in miniature (WIM) to the end user of the AR-System as kind of navigation help. To do this he needs several components that work together: A presentation service to render the WIM (Viewer), a service that provides the position of the user (Tracker) and several services to filter the position data (Filters). First Maxi needs to start the Services either by using the shell, in case he needs the console output or by using DIVE. Now that the Services are running, Maxi changes the Attributes and Predicates of the Services to fit in his application. This also includes connecting Services via DIVE. When everything is running, the essential part begins: the fine-tuning. Every filter service has special features that need to be tuned. Maxi uses DIVE to open a costum, filter specific configuration tool to adjust several parameters. As soon as Maxi is happy with the configuration he makes the ServiceDescriptions persistent by saving them to XML-files.

Configuring and monitoring a Service

The developer Fabi[3] wants to build a new component similar to one that already exists. He starts DIVE and selects the service, he wants to use as blueprint. He opens a dialog with the XML-Description of this services and uses it to change it according to his ideas. After this he saves the description and starts to implement the service. When the service is ready to run Fabi used DIVE to monitor and fine tune its behavior. The color scheme and a list view of the services help him to find his service among all the existing ones. He edits the attributes and predicate checks whether they work fine with the rest of the system. After Fabi is happy with his configuration he saves the service description to make his work persistent.

Showing the System to an Audience

The CAR Team presents their work to an external audience. In order to explain the software architecture, that drives their presentation, they use DIVE to show a graph view of the service network they built. Now they want to make clear, that the system is distributed over many hosts. To do this, they group the Services by their host name Attribute. Now the audience sees the data flows between different hosts. To reduce complexity, the graph view is colored to distinguish the status of Services and Services of the Middleware can be hidden.

[2]The nickname of one of the CAR developers
[3]The nickname of another CAR developer

2.2.2 Actors

What Actors can be extracted from the scenarios? The Actors have mainly stayed the same since the initial work, with one exception: The actor `User` has been differentiated. We now have two different types of them ...

User The `User` stands for the developer, that uses DIVE in his daily work. It is also the one, who uses DIVE during presentations to explain the DWARF to `Visitors`. As seen in the scenarios, we further distinguish two types. Please note that every time the general term `User` is used, both of them are addressed.

- **Application Developer** `ApplicationDeveloper` concentrates on the whole application rather than on details of the components. His task is to combine and configure components to form one application. The extensions designed for this kind of user aim to support this kind of authoring work.
- **Component Developer** He concentrates on the design of single components. He does not need a high level manipulation of the whole system. Tuning single components and configuring their interfaces is the focus of his work.

Visitor The to `Visitors` does not work with the DWARF directly. He rather wants to get an overview and thus need information about DWARF in reduced complexity.

DWARFServiceManager While the first two Actors are of flesh and blood, the `DWARFServiceManager` is not. The `DWARFServiceManager` is part of the DWARF middleware. In the original work of Daniel Pustka, he only served to retrieve information about the DWARF system. Now, his interface is also used to commit changes in the running DWARF system.

2.2.3 High Level Use Cases

The following three use cases are high level use cases and directly derived from the scenarios. The use cases `ProgramStartup` and `ProgramShutdown` that are included by them, have not changed and are thus not repeated here. They can be found in [16].

Configure Services

Participating actors Initiated by `ComponentDeveloper`

Entry condition
1. The `ComponentDeveloper` starts DIVE (includes `ProgramStartup`).

Flow of events

2. The `ComponentDeveloper` selects the `Service` that builds up his component. (includes `Select Service` or `Selecting a Service via ListView`)
3. He now starts to change them. This includes `Changing the Attributes` and `Change Predicates`.
4. When the `Service` suits him, he makes the changes persistent. This includes `Make Changes Persistent`

Exit condition

5. The `User` closes DIVE (includes `ProgramShutdown`).

Configure Application

Participating actors Initiated by `ApplicationDeveloper`

Entry condition

1. The `ApplicationDeveloper` starts DIVE (includes `ProgramStartup`).

Flow of events

2. The `ApplicationDeveloper` uses the `GraphView` to get an overview. This is supported by coloring and grouping the Services.
3. The `ApplicationDeveloper` selects one of the `Services` of her Application. (includes `Select Service` or `Selecting a Service via ListView`)
4. She now performs changes. These may include:
   - `Change Attributes` and `Changing the Predicates`.
   - `Connect Services`
   - `Start Services`
   - `Configure Services`
5. When the `Service` suits her, she makes the changes persistent. This includes `Make Changes Persistent`, `Change Attributes` and `Change Predicates`.
6. Steps 3 and 5 may be performed as often as it takes to fully configure the application.

Exit condition

7. The `User` closes DIVE (includes `ProgramShutdown`).

Demonstrate System Design

This use case describes how DIVE is used by a `User` in order to demonstrate a system design to a `Visitor`. This is an extended Version of the original use case.

Participating actors Initiated by `User`
`Visitor`

Entry condition

1. The `User` starts DIVE (includes `ProgramStartup`).

Flow of events

2. The `User` explains the system design to a `Visitor`. To show details about a service, the `User` clicks on a node in the diagram (includes `SelectService`).
3. When the `User` has changed the system (e.g. by starting a new service), DIVE automatically updates the graph to show the changes (includes `UpdateView`).
4. to demonstrate the connections between different service groups, the `User` enables the grouping option to group the services based on one of their `Attributes`. (Figure 2.2 shows the grouped view from the final implementation.)
5. The `Visitor` may formulate change or configuration requests that can be conducted live by the `User`. This includes `Configure Application`.
6. The `User` continues with steps 2 to 5.

Exit condition

7. The objectVisitor is happy with the knowledge gained during the demonstration and the `User` closes DIVE (includes `ProgramShutdown`).

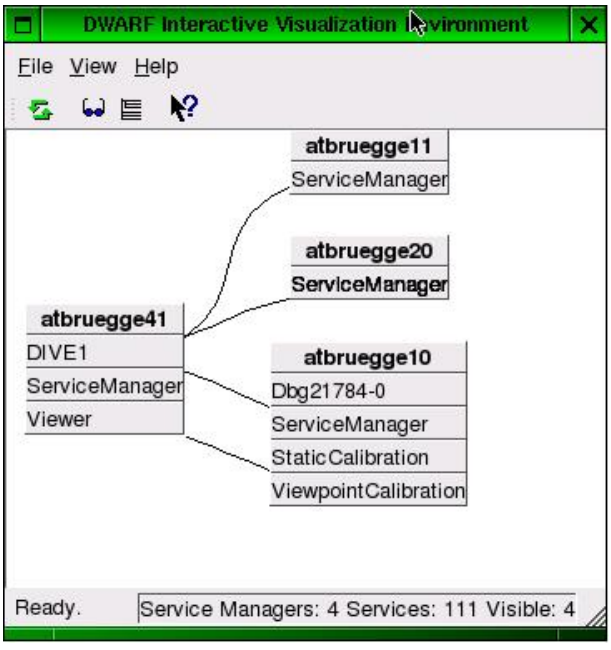


Figure 2.2: The GUI of a grouped view from the final implementation.

2.2.4 Low Level Use Cases

The rest of the use cases describe system behavior of a lower level. Most of them are included by the high level use cases. Furthermore, all of these use cases are based on the assumption that the application DIVE is already running. They thus may include the use case `ProgramStartup` and maybe `ProgramShutdown` if they are not seen in the context of the high level use cases. For clarity's sake these details are omitted here. The use cases documented in [16] which are used here include ...

1. `Update View`
2. `Select Service`. This use case will be reformulated in order to adapt to the new use case `Finding Services with the List View`
3. `Demonstrate System Design` Also this use case will be reformulated in order to take into account the new possibility to group `Services` in the `GraphView`

Select Service via the `ListView`

This use case describes how a user gets additional information about a service, in an extended version of the original, as it was mentioned before. Figure 2.3 shows the `ListView`.

Participating actors	Initiated by `User`
Entry condition	1. The application has started and the model is updated.
Flow of events	2. The `User` opens the `ListView`. 3. A list of all `Services` is displayed, grouped by attributes valuable for the decision making of the `User`. 4. The `User` browses the list. He may use the sorting feature to speed up his search. 5. The `User` has found his `Service` may now decide whether to select the `Service` itself of one of its `Needs` or `Abilities`.
Exit condition	6. A new `PropertiesWindow` becomes visible. It shows a list of all attributes of the selected `Service` or its `Needs` or `Abilities`.
Special requirements	There is always at most one `PropertiesWindow` visible. If there already is one, it will be re-used for the new `Service`.

Change Attributes

This use case describes how DIVE is used by a `User` in order to change `Attributes`. Figure 2.4 shows the dialog of the implementation.

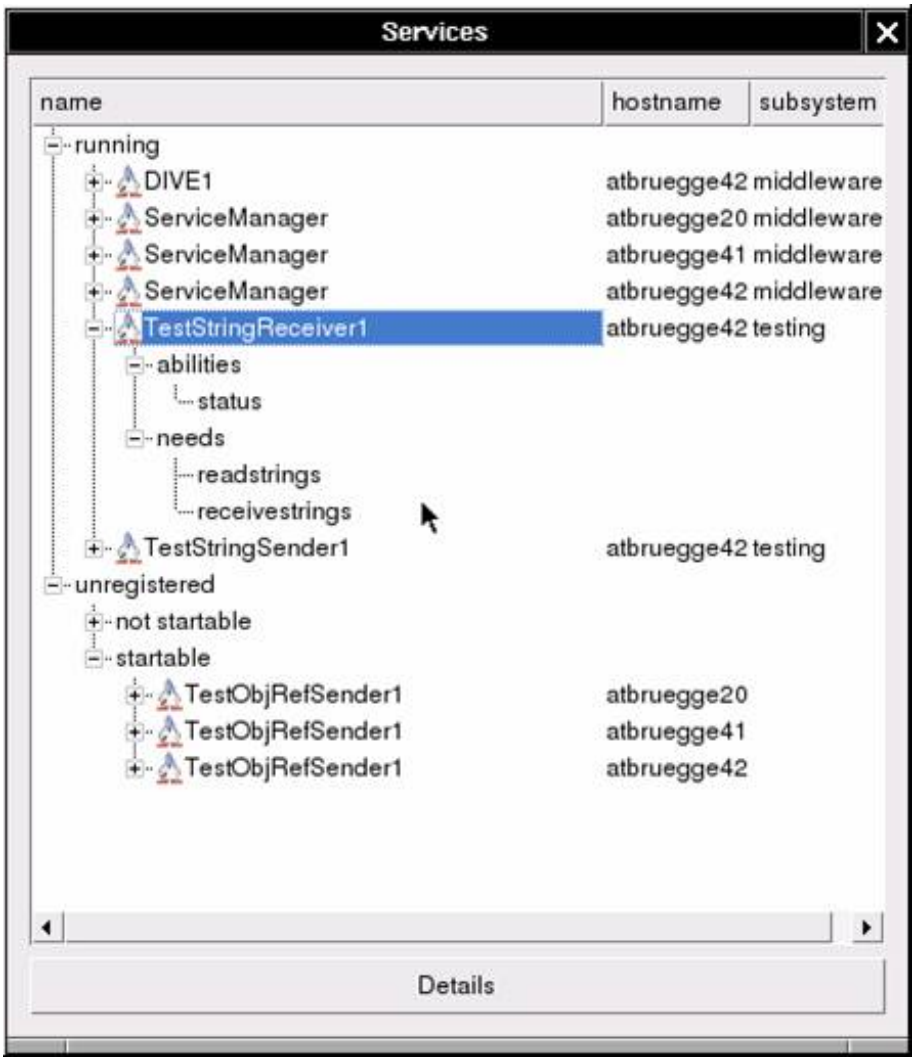


Figure 2.3: The GUI of the `ListViewDialog` from the final implementation. The user may browse through the services. He is supported by the the sorting of services, based on several attributes.

Figure 2.4: Dialog for changing Attributes as used in the final implementation.

Participating actors	Initiated by `User` `DWARFServiceManager`
Entry condition	1. The `User` selects an `Ability` of a `Service` (includes `SelectService` and may also include `Finding Services with the List View`) and chooses to edit Attributes.
Flow of events	2. The `User` is presented a list of all Attributes where read-only Attributes are disabled 3. The `User` edits Attributes or creates new ones. This step may repeat several times 4. The `User` confirms the changes 5. DIVE orders the `DWARFServiceManager` to commit the changes. 6. The view is updated (includes `Update View`) to reflect the changes.
Exit condition	7. The `User` sees his changes in the newly updated `PropertiesDialog` including the induced changes of the system.

Change Predicates

This use case describes how DIVE is used by a `User` in order to change `Predicates`. Figure 2.5 shows the dialog of the implementation.

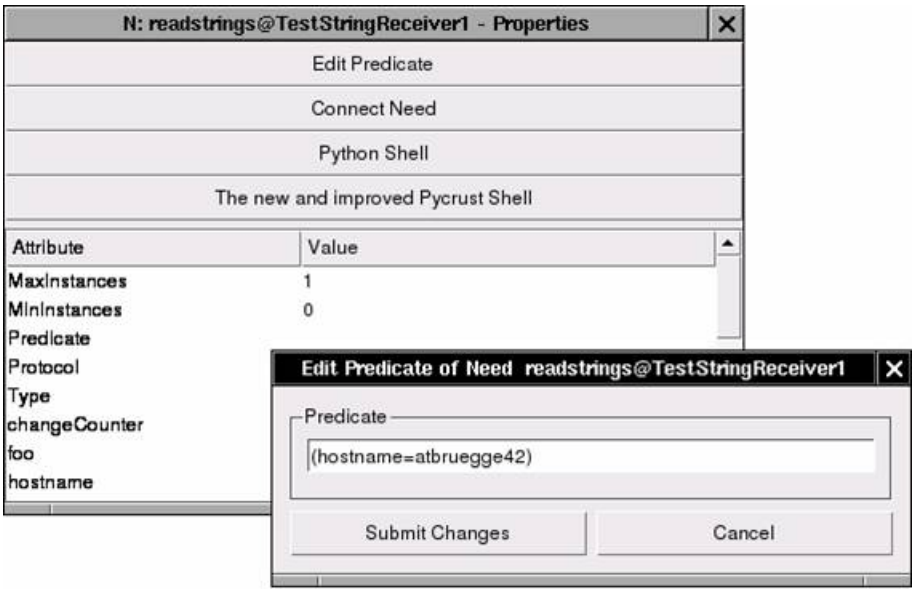


Figure 2.5: GUI of the extension "Changing Predicates" as used in the final implementation.

Participating actors	Initiated by `User` `DWARFServiceManager`
Entry condition	1. The `User` selects `Need` of a `Service` (includes `SelectService` and may also include `Finding Services with the List View`) and chooses to edit Predicates.
Flow of events	2. The `User` is presented a Dialog with the Predicate 3. The `User` edits the Predicate 4. The `User` confirms the changes 5. DIVE orders the `DWARFServiceManager` to commit the changes. 6. The view is updated (includes `Update View`) to reflect the changes.
Exit condition	7. The `User` sees his changes in the newly updated `PropertiesDialog` including the induced changes of the system.

Start Service

This use case describes how DIVE is used by a `User` in order to change start Services.

Participating actors	Initiated by `User` `DWARFServiceManager`
Entry condition	1. The `User` has found an inactive `Service` he wants to start. This may include the use case `Finding Services with the List View`
Flow of events	2. The `User` selects this `Service` (includes `SelectService`) 3. The `User` chooses the Extension to start `Services` 4. DIVE changes the `ServiceDescription` of this `Service` in a way that activates the auto start mechanism of the `DWARFServiceManager` 5. The `DWARFServiceManager` activates the `Service`. 6. The view is updated (includes `Update View`) to reflect the changes.
Exit condition	7. The `User` sees the newly started `Service`.

Connect Services

This use case describes how DIVE is used by a `User` in order to connect the `Need` of a `Service` to one or more Abilities of other `Services`. Figure 2.6 shows the dialog of the implementation.

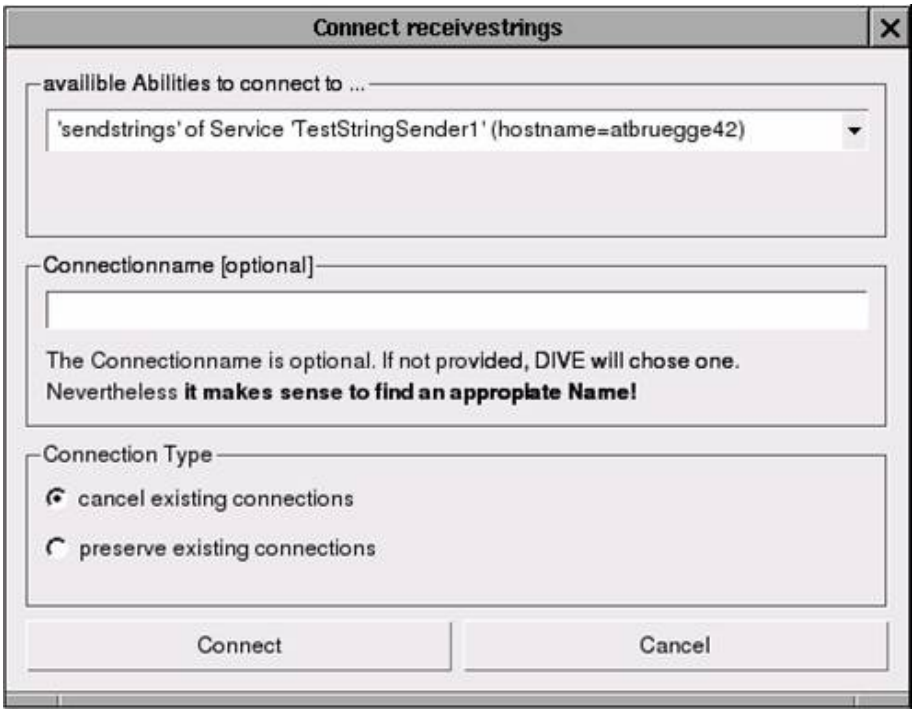


Figure 2.6: Dialog for connection establishment from the final implementation.

Participating actors Initiated by `User`
`DWARFServiceManager`

Entry condition

1. The `User` has found a `Need` of a `Service`, he wants to connect. This may include the use case `Finding Services with the List View`

Flow of events

2. The `User` selects this `Service` (includes `SelectService`)
3. The `User` chooses to connect `Services`
4. The `User` is presented a list of fitting Abilities he is able to connect to
5. The `User` chooses the `Ability` he wants to connect to.
6. The `User` is asked to choose an attribute value the connection is identified with. This attribute is used by the predicate to reference the connection. It can be seen as a connection name.
7. The `User` enters a value or leaves finding a value to DIVE
8. `DIVE` asks the `User` whether existing connections with this `Need` should be preserved. Fomally the question is whether to establish a 1:1 (connections not preserved) or a 1:n connection (connections preserved).
9. The `User` decides or just uses the default.
10. The `User` confirms the dialog.
11. `DIVE` uses the interface of the `DWARFServiceManager` to change the `ServiceDescription` of both `Services`
12. The view is updated (includes `Update View`)

Exit condition

13. The `User` sees the newly formed connection.

Make Changes Persistent

This use case describes how DIVE is used by a `User` in order to make his changes of the `Services` persistent. Figure 2.7 shows the dialog of the implementation.

Participating actors Initiated by `User`
`DWARFServiceManager`

Entry condition

1. The `User` selects a `Service` (includes `SelectService` and may also include `Finding Services with the List View`)

Flow of events

2. The `User` chooses the Extension to save `ServiceDescriptions`.
3. DIVE queries the `ServiceDescriptions` in XML from the `DWARFServiceManager`.
4. The `User` is presented a dialog with the `ServiceDescriptions` in XML.
5. The `User` may now change this `ServiceDescriptions` in XML as he likes.
6. The `User` chooses to save it.
7. DIVE presents the `User` a `SaveFileDialog`.
8. The `User` chooses an appropriate name plus directory and confirms.
9. DIVE saves the `ServiceDescriptions` in XML to the choosen file.

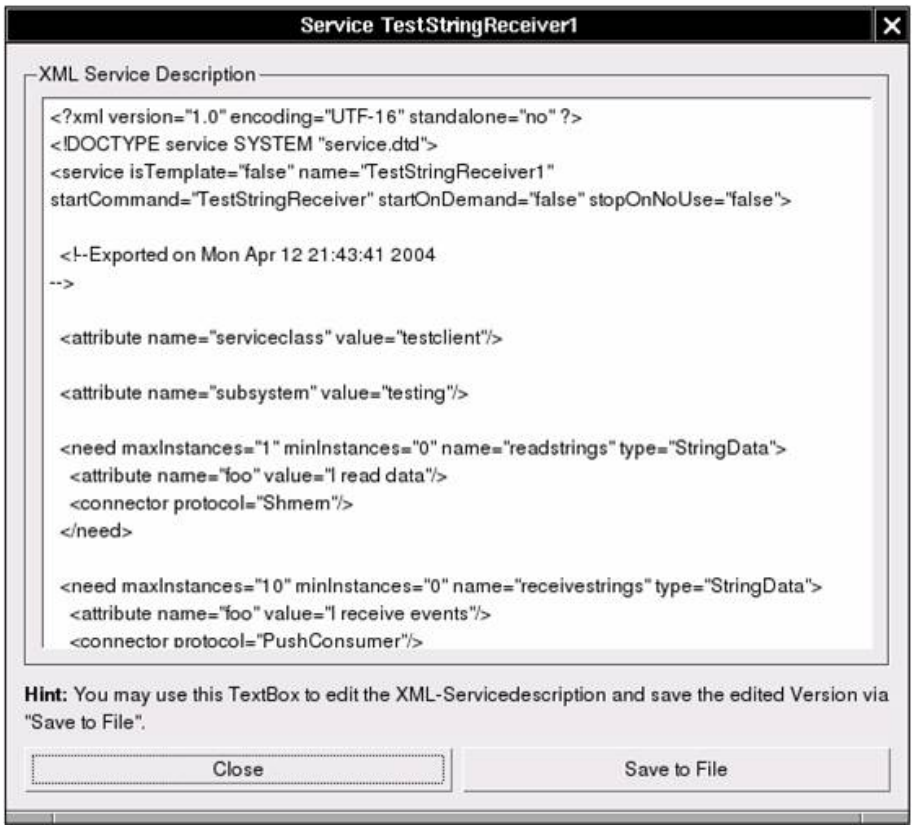


Figure 2.7: Dialog for saving the ServiceDescription in XML as used in the final implementation. The user may use this dialog to edit the XML code before saving it.

Exit condition 10. The `User` is able to see the newly created file in his favourite file browser.

Configure Services

This use case describes how DIVE is used by the `ApplicationDeveloper` to configure `Services`. Figure 2.8 shows the GUI of a sample `Configurator`.

Participating actors Initiated by `ApplicationDeveloper`

Entry condition 1. The `ApplicationDeveloper` selects an `Ability` of a `Service` (includes `SelectService` and may also include `Finding Services with the List View`) and chooses to use the Extension to configure this `Service`.

Flow of events 2. `DIVE` retrieves the appropriate `Configurator` to configure the `Service` and activates it.

3. The `ApplicationDeveloper` makes the configuration changes. (how changes are made is left to the particular `Configurator`)

Exit condition 4. the `Configurator` runs, is connected to the service and sends configuration data.

2.3 Object Model

The objects that where identified during the requirements elicitation split into two groupes. The following list also includes objects that were already found, documented and implemented by Danial Pustka. So the list is a brief repetition as well as a brief summery of newly found objects:

Entety Objects belonging to DIVE Most Entety Objects used by DWARF have a counter part in DIVE. They are used to cache the data retrieved by the CORBA references, these are italicized in the list. Additionally there are some Entety Objects only used in DIVE. The most important ones are:

- DWARFSystemModel
- ServiceManagerSession
- *DWARFServiceManager*
- *Service (Counterpart to the ServiceDescription in DWARF)*
- *Need*
- *Ability*

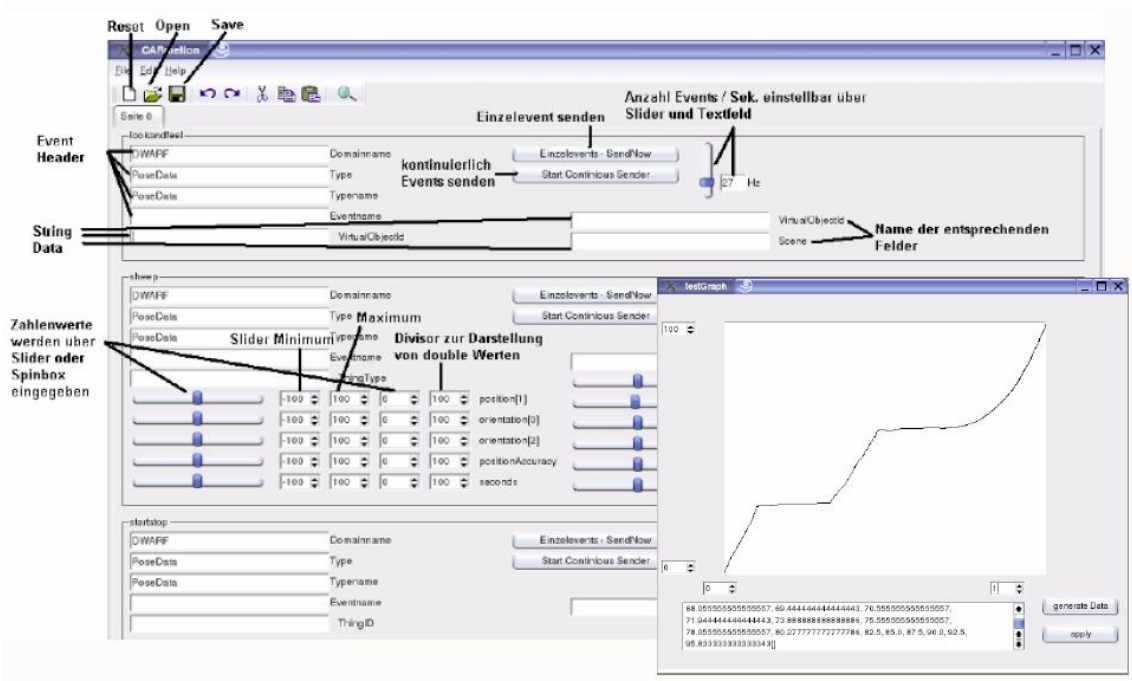


Figure 2.8: A GUI of an example configurator used in the CAR-Project. This picture is taken from [9] and shows a rich set of widgets to generate configuration data for the filter services used in CAR.

- *Attribute*

Boundery Objects include all elements of the graphical user interface especially the dialogs. The most important ones are:

- ListView and ListViewDialog
- GraphView
- PredicatesDialog
- AttributesDialog
- ConnectServiceDialog
- XMLDescriptionDialog
- SaveFileDialog

Chapter 3

System Design

> "There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies."
> *C. A. R. Hoare*

We will now have a look at the system design of DIVE. The purpose of this chapter is to give a rough overview. The points that changed compared to the former version of DIVE will be discussed.

3.1 Design Goals

In the original work on DIVE three main design goals were stated. Which of them have changed?

Reusable components It was stated: "All components of DIVE should work independently of each other, so they can be re-used easily in future projects. This also results in a clean design with clear responsibilities." But this goal was not fully achieved: The class `DIVEApp` included code for updating The `GraphView` and dialogs. But according to the object oriented paradigm it is the `GraphView` and the dialogs that are responsible for their own update procedure. Thus this design goal is of unaltered importance.

Extensibility This is a point that became even more important. In the original work it mainly aimed towards the extension mechanism for the "debuggers". Now this is not enough: It is the case, that parts of DIVE need to be extended, that nobody thought about. While for the "debugger" extensions it was the ideal decision to build an extension mechanism, but it does not make sense, to build extension mechanisms for everything. The best way to at least provide an easy way to further extend DIVE is to enforce the first design goal: "Re-usable components" with the point "clear responsibilities" stressed.

Compatibility with DWARF This point has lost importance. Not that the compatibility is not needed anymore. The point is: This design goal has already been reached. DIVE is perfectly integrated into the DWARF-System.

Are there any new design goals? Yes, at least one:

Focus on the user Now, that DIVE is already a fully functioning application. It is neccessary to focus on the user. It is important to integrate extensions into DWARF, that are requested by him, to make the user interface more convenient to use, and so on. In the end it is the user who decides whether DIVE is successful or not. These concerns are especially met by co-operating with the CAR-Team including a final feedback which is discussed in chapter 5.

3.2 Subsystem Decompsition

We will now have a look at the major changes. A detailed view will be given in chapter 4. Figure 3.1 shows the over all class model of DIVE. The changes are marked in gray. Also the different subsystem can be seen. In order to discuss the changes, every subsystem will be addressed separately.

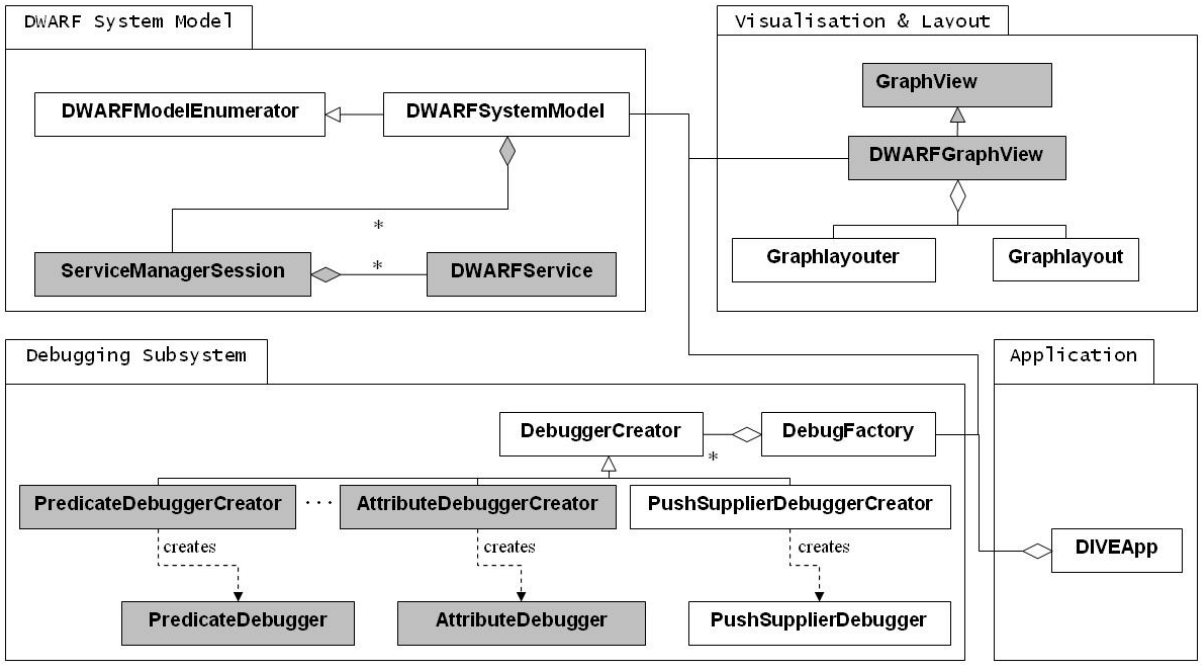


Figure 3.1: The new Classmodel, changes are draws in gray. The "..." is to be interpreted as a placeholder for many more extentions that where to space consuming to be shown.

3.2.1 DWARF System Model

The class model structure of this subsystem has undergone major changes, whereas the interface for the outside has hardly changed: First of all the routine for updating has changed fundamentally. The update is now iterative, so the model is not thrown away for each update. The update is also multi-threaded to minimize the waiting time due to network round trips for the Corba calls. Last but not least, version counting has been introduced to further speed up the update.

Besides the update routine, a new way of querying services from the model has been implemented: grouped querying. It is now possible to request the services grouped by one of their attributes. This feature is used by the graph visualization to provide a grouped view of the services.

3.2.2 Graph Visualization

In the graph visualization subsystem, a new abstraction layer has been introduced to separate the general functionality of a graph view from the more specific one of a graph view showing especially DWARF service networks. Furthermore a new way of displaying service networks has been implemented: a grouped view.

3.2.3 Application

Changes in this subsystem were mainly kinds of refactoring: The size in lines of code of the main application class has been reduced to enforce clarity and clean object oriented design. Also some changes have been made concerning the graphical user interface (GUI). A button bar makes it easier to access vital functions of DIVE and the new list view dialog simplifies the search for specific services.

3.2.4 Debugging

The structure of this subsystem has not been changed, rather the existing extension mechanism has been used to plug in a variety of new features. These will be described in detail in section 4.4. Most of these extensions directly address a specific use case, discussed in 2.2.

Chapter 4

Object Design

What is true for the same chapter in the predecessor work of Danial Pustka [16] is also true for this chapter ...

> "This chapter is rather technical and intended for future developers who wish to change or extend DIVE."

Further more it is assumed that the reader is familiar with the predecessor work because this chapter will only discuss changes and extensions and will not repeat the original object design. We will start our discussion with a list of objects that had to be modified. It will be explained how they have been changed and what the rational, this change is based on, was. This is followed by an explanation of the new update procedure and the list of extensions.

4.1 Refactored Objects

4.1.1 DwarfSystemModel and ServiceManagerSession

This object is used to retrieve and hold all information about the DWARF system. This includes all cached information of all Services as well as CORBA references to the running ServiceManagers. Two major changes had to be applied to the DwarfSystemModel:

Adding CORBA references of Services, Needs and Abilities

So, DwarfSystemModel includes a list of DWARFServices that is used to cache information because CORBA calls are time consuming. There is only one problem: If you want to change the system, interact with it, changing cached information will not do. You have to get a CORBA reference to the Service, you want to manipulate. But where to get one? The first possibility is finding the Service based on his ID and the host name. But that is rather complicated: You have to get a connection to the responsible ServiceManager based on the cached host name, then retrieve the Service based on its ID. Last but not least

the Need or Ability that is due for manipulation has to be retrieved based on its name. The second possibility would be to copy all CORBA references that are used during the update procedure and store them right away with the other information. Of course these references may become invalid, but as this can happen all the time, code to check this has also be included in the first mentioned approach.

Rationale: It was decided to use the second method because it yields better performance and easier implementation of the extensions. The overhead of storing space can be neglected.

Giving the responsibility for Services to ServiceManagerSession

In the original design, the list of Services was held in the DwarfSystemModel. This may sound convenient because all services can easily be retrieved by one call on the `DwarfSystemModel`, namely `getService(string name)`. But this does not reflect reality: In DWARF ServiceDescriptions are not held by a central instance. According to [10] it is always a good idea to write code that reflects basic structure of the application domain as close to reality as possible.

The ServiceDescriptions are held by the `ServiceManagers` which are represented by the `ServiceManagerSessions` in DIVE. To preserve the comfort of uniformly querying all `DWARFService`-objects from the central `DwarfSystemModel`, `DwarfSystemModel` holds a central list with all the service ids as strings. When a `DWARFService` is queried, this query is delegated to the responsible `ServiceManagerSessions`. The complex structure is thus hidden behind a facade according to the facade-pattern [7]. All existing objects need not to be changed.

Besides the higher correlation with the DWARF reality, moving the responsibility for the ServiceDescriptions to the ServiceManagerSessions yields another advantage: It enables multi-threaded updates because the ServiceManagerSessions are independent from each other. We come back to this point in the section about the new update procedure, section 4.2 on page 31.

4.1.2 DIVEApp

`DIVEApp` is the main class of DIVE and is derived from QT's `QMainWindow`. It is intended to host the glue and control logic of the application. Unfortunately the class also hosted code to build up the `GraphView` and to update certain dialogs, code that should be encapsulated in the corresponding classes. This is exactly what was done to reduce the size of `DIVEApp`. But the process is not finished yet. In order to make `DIVEApp` really clean, even more refactoring effort should be invested as we will see in the chapter about future work, chapter 6 on page 45.

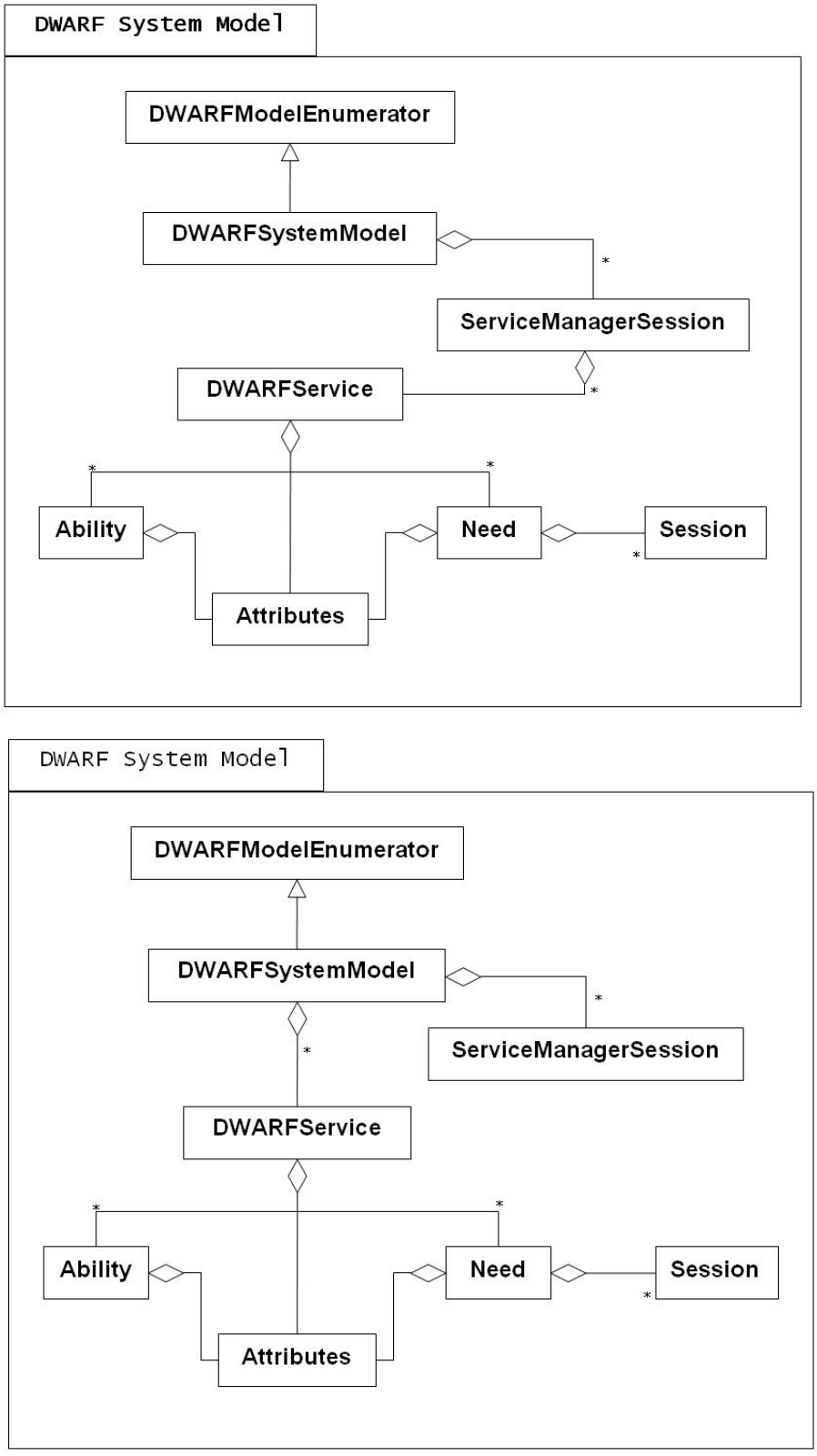


Figure 4.1: The new relationship (up) between DwarfSystemModel, ServiceManagerSession and the Services compared to the old one (down).

4.1.3 GraphView

The above mentioned refactoring of `DIVEApp` made it necessary to restructure `GraphView`. `GraphView` encapsulates all functionality to build network graphs. But the specific graph used in DIVE, the graph that is built, based on the information in `DWARFSystemModel` was constructed in `DIVEApp`. As I already mentioned this is not the right place to do so. But a general class to build graphs like `GraphView` is not the right place either. I decided to introduce a new class `DWARFGraphView` that takes responsibility for the construction of the graph. `DWARFGraphView` is derived from `GraphView` as shown in Figure 4.2. This change is also neccesary for another feature: A graph view with groups of services as nodes. The grouped graph view is discussed in section 4.3 on page 34.

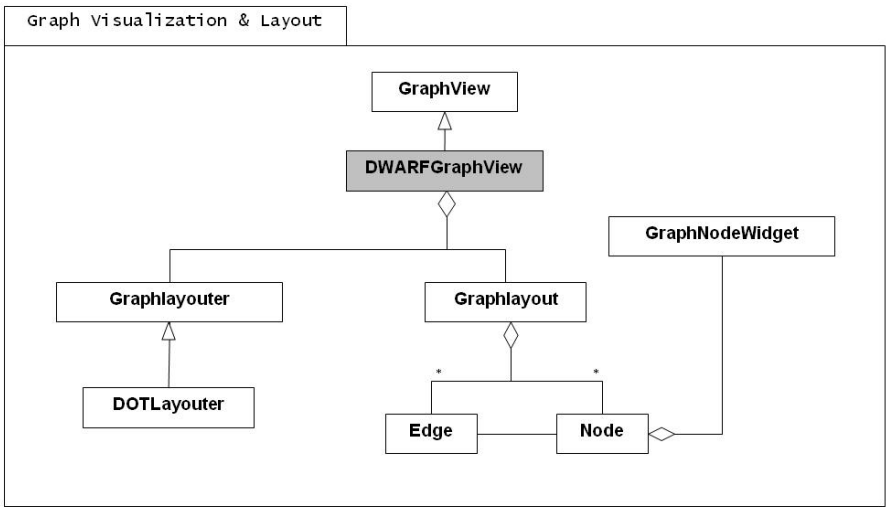


Figure 4.2: The new relationship (up) between DwarfSystemModel, ServiceManagerSession and the Services compared to the old one (down).

4.2 The New Update Procedure

Analysis of the old update procedure yielded that not the exact amount of transmitted data slows down the update, but the number of CORBA calls. So there are two ways to speed things up:

1. Reducing the number of CORBA calls per update
2. Reducing the time, the application is stalled per CORBA call.

This insight resulted in two independent measures. But before the realization of these, same other changes had to be made: The old update routine was built upon the "throw away and build new from scratch" principle. The simple this method was, it did not allow

iterative updates that pioneer the way to more sophisticated methods. The new iterative update consists of three stages:

1. Iterate through our list of `Services` and mark all as "not up to date". The mark is realized as a property of the class `Service`.

2. retrieve one `Services` from the `ServiceManager`. If the `Service` is already in our list, update it, if not insert it. After this mark the Service as "up to date". Repeat this until all `Services` are retrieved.

3. Iterate through our list of `Services` and delete all that are marked "not up to date".

Please note, that this iterative update will not yield a measurable increase in speed. As mentioned before it only sets the stage for the following methods that do yield remarkable higher speed.

4.2.1 Version Counting

Version counting aims to reduce the calls per update. The idea behind this, is the fact that in between two updates the world does not change that much. `Services` that did not change do not need to be retrieved. But how to know that one `Service` did not change? The key to success is to introduce a version counter. This counter is maintained by the `Service Manager` as an attribute of each service, and called "changeCounter". Only if the newly queried version has changed, the `Service` has to be updated. The logic for version checking is encapsulated in the method `Update()` of `DWARFService`. By doing this, the procedure is transparent and no line has to be changed in any other class.

4.2.2 Multithreaded Update

The CORBA call used to retrieve data about services only return small amounts of data: often only one CORBA reference. So most time during the call is consumed by the network round trip time. In the old version of DIVE, the application was stalled during this time, or, more exactly, the update thread. But as the whole application had to wait for the new information the update thread provided, this did not make a difference. So the idea is to use this time to initiate the next calls. As was mentioned in section 4.1.1 on page 28 the refactoring of the `DwarfSystemModel` helps us here. As the `Services` are held by the `ServiceManagerSessions`, which are independent from each other, it is quite obvious to make each `ServiceManagerSessions` a Thread by its own. This thread runs in an endless loop. And this is what it does in this loop:

1. `ServiceManagerSessions` calls its own method `Update()`.

2. `Update()` performs the action described at the beginning of section 4.2 on the page before.

3. The central service list in `DWARFSystemModel`, containing only the ID-stings, is synchronized in every step. Hazards are ruled out by semaphores.

4. Finally, `ServiceManagerSessions` waits for a `QWaitCondition` that is also known to `DWARFSystemModel`.

5. After being woken up, the loop repeats.

The wake-up call is performed by the method `Update()` of `DWARFSystemModel`. Figure 4.3 shows an overview.

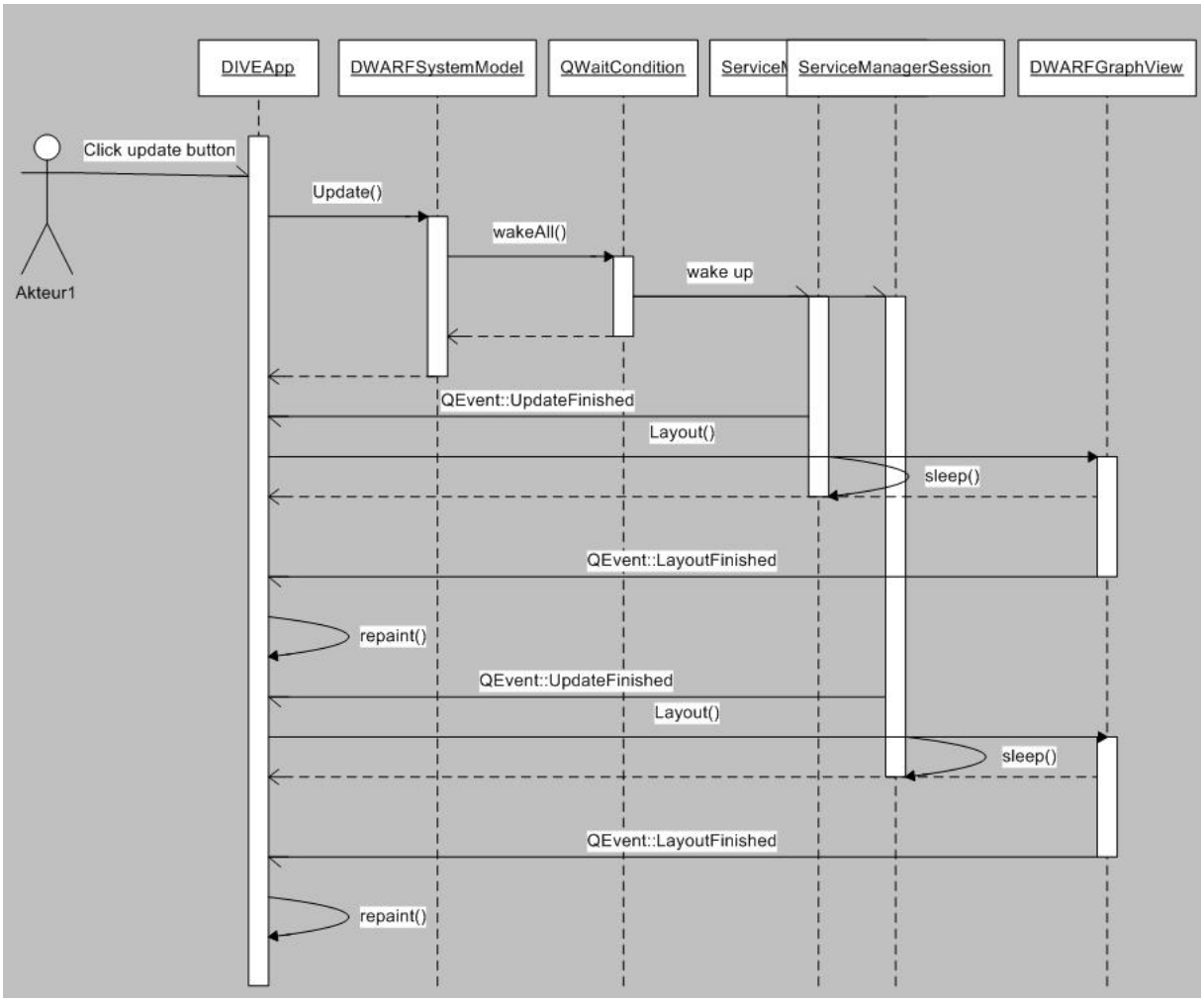


Figure 4.3: Sequence diagram of the multi-threaded update. Please note, that in this example there are only two ServiceManagerSessions, but there may be arbitrarily many of them.

All this has also a beneficial side effect: Formally the `DIVEApp` was responsible for creating the update thread, which further complicated it. With the new approach `DIVEApp` has only to call `Update()` upon his reference to the instance of `DWARFSystemModel`.

4.3 New Views

Two new views on the `DWARFSystemModel` have been added to DIVE: One of them is, the grouped graph view is rather a feature, extending to the graph view. The second is a complete new view and comes in dialog form.

4.3.1 List View

The list view shows the `Services` as a list, that can be sorted by different criteria. The list view is realized as a `QDialog` containing a `QListView`. The GUI is shown in Figure 2.3 on page 16. Some `QListViewItems` do not contain service information, they serve as grouping items. The `Services` can be grouped by their activation status or and their auto-start property. All the logic is included in the class: `ListViewDialog`. An instance of `ListViewDialog` is held by `DIVEApp`.

4.3.2 Grouped Graph View

In order to provide a grouping functionality, the class `DWARFGraphView`. A method for laying out the standard graph already exists. So, the easiest way is to add a new method that provides grouped layout. The grouping is done based on an attribute. Grouping Details. Figure 2.2 on page 14 shows a screenshot of the grouped view with grouping attribute "host name".

If more other layout strategies are planned to be implemented in the future, it would be worth thinking about a strategy pattern here. In this case however with only two simple methods and no intent to extend further, a strategy pattern is, in my opinion, not worth the overhead.

4.4 Extensions

The extensions all use the extension mechanism that was introduced by Daniel Pustka. Note that he used the notation "Debugger" for extensions. As I disagree with this naming[1], I will stick to "extension". Only for the class names, I will adopt the expression "Debugger" for I wish to hold the class names consistant with their super classes. As it was shown in Figure 3.1 on page 26, all extensions are derived from the objectDebuggerCreator.

So most of the extensions have two main parts that are also reflected in a separation in different source files:

1. One or two classes that host the logic and implement the extension functionality.
2. A GUI in form of a QT-Dialog which is generated mainly via Trolltech's GUI generator. (not all extensions use a GUI)

For all extensions the GUI is presented as Figure and the logic is described in written form.

[1] Most of the extensions do not satisfy the definition of a debugger. They are "just" extensions.

4.4.1 Changing Predicates

This extension implements the use case `Changing Predicates`. Three objects are used, the first two for the control, the last one for the GUI:

`PredicatesDebuggerCreator` ... implements the `DebuggerCreator` Interface. The button for this debugger is only shown for `Needs` of `Services`, that have the `Attribute` user name set to their login name.

`PredicatesDebugger` This class retrieves the CORBA reference to the `Need` and gets the `Predicate` in form of a STL `string`. It creates the `PredicatesDebuggerDialog`] and passes the string to it. The dialog is then presented to the `User`. If the `User` confirms his changes, the CORBA reference is used to set the new `Predicate`.

`PredicatesDebuggerDialog` Figure 2.5 on page 18 shows the GUI.

4.4.2 Changing Attributes

This extension implements the use case `Changing Attributes` Three objects are used, the first two for the control, the last for the GUI:

`AttributesDebuggerCreator` ... implements the `DebuggerCreator` Interface. The button for this debugger is only shown for `Abilities` of `Services`, that have the `Attribute` user name set to their login name.

`AttributesDebugger` This class retrieves the CORBA reference to the `Ability` and gets the `Attributes` in form of a STL key-value-list of `strings`. It creates the `AttributesDebuggerDialog`] and passes the list to it. After the `User` confirms changes he made in the dialog, the CORBA reference is used to overwrite the `Attributes` with the changes ones.

`AttributesDebuggerDialog` Figure 2.4 on page 17 shows the GUI. The dialog is responsible for showing an editable list of the `Attributes`. Every `Attribute`, that matches an entry in a black list of read-only Attributes is disabled. The `User` may now insert, delete and modify the entries as he likes, until he cancels or confirms the dialog.

4.4.3 Connection Establishment

This extension implements the use case `Connecting Services` Three objects are used, the first two for the control, the last for the GUI:

`AttributesDebuggerCreator` ... implements the `DebuggerCreator` Interface. The button for this debugger is only shown for `Needs` of `Services`, that have the `Attribute` user name set to their login name.

`AttributesDebugger` This class retrieves the CORBA reference to the `Needs` and gets the `DataType`, the `Need` requires as well the connector type. It then iterates through all the `Abilitess` of the running `Services` and adds all fitting ones to a list. This list is passed to `AttributesDebuggerDialog`. There after the `AttributesDebuggerDialog` is shown. When the `User` confirms the dialog, `AttributesDebugger` adds the `Attribute` "connectionName" to the chosen `Ability` and sets it to the value entered by the `User`. If the `User` refused to enter one, `AttributesDebugger` chooses one based on the `Need` and `Ability` names. The `Predicate` of the `Need` is set to accept `Abilities` with the accordant connection name. If preservation of existing connections is choosen, the new predicate is logically connected to the old one with an "or".

`AttributesDebuggerDialog` Figure 2.6 on page 20 shows the GUI. It presents the list of matching `Attributes` as a pull down list. It also contains a field for the connection name and a radio button to decide whether existing connection shall be preserved. The `User` may cancel or confirm the dialog.

4.4.4 Starting Services

This extension implements the use case `Starting Services`:

`StartDebugger` ... implements the `DebuggerCreator` Interface. The button for this debugger is only shown for inactive `Services` that have Autostart set to `true`. To start a `Service`, it performs the following steps:

1. Get the `Need` "servicestarter" of DIVE and set its `Predicate` to "(AND(serviceID=[SERVICEID])(abilityName=startMe))", where [SERVICEID] stands for the ID of the `Service`, that is to be started.
2. Add the Ability "startMe" to this `Service`. Set its Connector to "Null" and its type to "Start".
3. The `ServiceManager` will now start the `Service`.

4.4.5 Configuring Services

This extension implements the use case `Configuring Services` Two objects are used for control:

`FilterConfigurationDebugger` ... implements the `DebuggerCreator` Interface. The button for this debugger is only shown for `Services` that have the `Attribute` user name set to the `User's` login name. Figure 4.4 on the next page shows the steps that are performed to open a configuration service. Here is the explanation:

1. DIVE reads the `ServiceDescription` of the `Service` to be configured.

2. A new `ServiceDescription` is generated by DIVE. It is named "ConfiguratorOf[SERVICENAME]", where [SERVICENAME] is the name of the `Service`, that is to be configured. The `ServiceDescription` gets the `Attribute` "ProvidingDataFor" set to [SERVICENAME]. "StartOnDemand" is set to `true`, "StopOnNoUse" to `true` and "StartCommand" is set to the value of the service's "configuratorExecutable"-attribute. For every `Need` with the `Attribute` "ConfigurationNeed" set to `true`, a new `Ability` with the same name and data type is added. The Predicate of the corresponding `Need` is set to (AND(ProvidingDataFor=[SERVICENAME])(abilityName=[NEEDNAME]))"
3. The `ServiceManager` will now start the configuration service with this `ServiceDescription`.
4. The configuration service has to read its own `ServiceDescription` and adjust its user interfaces.
5. The configuration service can now use its connection to the service to be conficured to send configuration data.

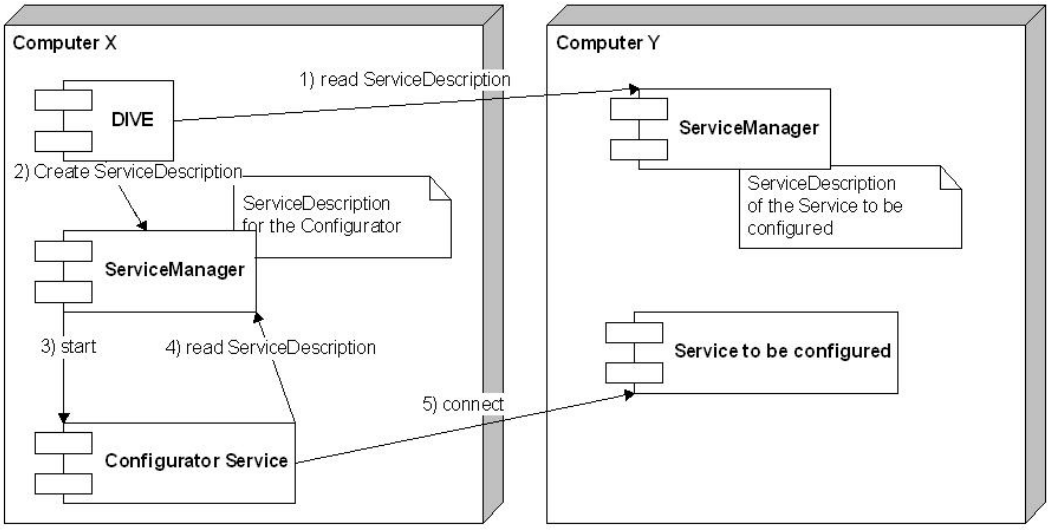


Figure 4.4: The steps that are performed to configure a service.

4.4.6 Saving the XML Description

This extension implements the use case `Making Changes Persistent` Two classes are used:

`GetXMLDescriptionDebugger` ... implements the `DebuggerCreator` Interface. The button for this debugger is shown for every `Service`. The `GetXMLDescriptionDebugger` uses the CORBA reference of the selected `DWARFService` to query the

`ServiceDescription` in XML form. This string is then passed to the `GetXMLDescriptionDebuggerDialog`.

`GetXMLDescriptionDebuggerDialog` This dialog just presents the passed string in a `QEditBox`. By clicking the "Save to File"-button, a save dialog appears. If it is confirmed, the string is saved to a file with the extension "xml". Figure 2.7 on page 22 shows the GUI.

4.5 Other Changes

In this section two smaller additions to DIVE will be discussed, that are not directly related to the before mentioned objects.

4.5.1 Hiding Services of the Middleware

As DIVE is a monitoring tool that shows all services in the DWARF-Network, it also shows the services of the middleware: mainly DIVE itself and the service managers. Most users however are not interested in monitoring the middleware. So the filtering of Services already implemented in DIVE has been extended to also filter out middleware services. Three classes have been changed:

`DIVEConfiguration` The users choice whether to filter the middleware out or not is held persistent in this class.

`DWARFFilter` The actual filtering is implemented here.

`FilterForm` This dialog has been extended to hold a checkbox for the user to make his choice: to filter or not to filter.

4.5.2 Exporting of the Layout Source Code

In order to lay out the graph view of the services a layouter is used. So, DIVE produces input code for the layouter, the layouter gives back position information which DIVE uses for displaying the graph. But the layouter can also be used without DIVE: it can for example produce png-images from layout source code. So, it is quite obvious that the possibility of exporting the layout source code would help users to produce high quality pictures of the service network, for example to be used in documentations and publications. The implementation includes changes in two classes:

`DIVEApp` The pulldown menu has been extended to host a menu point named "export dot".

`DotLayouter` ... has now a new method to do the exporting.

Chapter 5

Results

> "I never made a mistake in my life; at least, never one that I couldn't explain away afterward."
> *Rudyard Kipling*

5.1 Update Speed Improvement

To measure the improvement of the update speed, I conducted a small experiment including the following four hosts in the DWARF network:

atbruegge22 including 1 running DIVE, 1 runnning ServiceManager and 64 inactive service descriptions.

atbruegge9 including 1 runnning ServiceManager, 10 running Services (TestStringSender) and 64 inactive service descriptions.

atbruegge10 including 1 runnning ServiceManager, 10 running Services (TestStringSender) and 64 inactive service descriptions.

atbruegge42 including 1 runnning ServiceManager, 10 running Services (TestStringSender) and 64 inactive service descriptions.

I modeled three different situations, that where used to test the update speed using the old and the new update algorithm. The situations where tested in the following order:

Situation A atbruegge22 and atbruegge9

Situation B Situation A plus atbruegge42

Situation C Situation B plus atbruegge10

Figure 5.1 shows the measurements in seconds. The time is counted until all information of all hosts is retrieved. But using the new algorithm, as soon as information retrieval of

one of the hosts is completed, this information is presented to the user. So the user gets information long before the update is completed. This is illustrated for situation C in Figure 5.2. In fact some hosts are finished rather quickly while others take much more time to be queried.

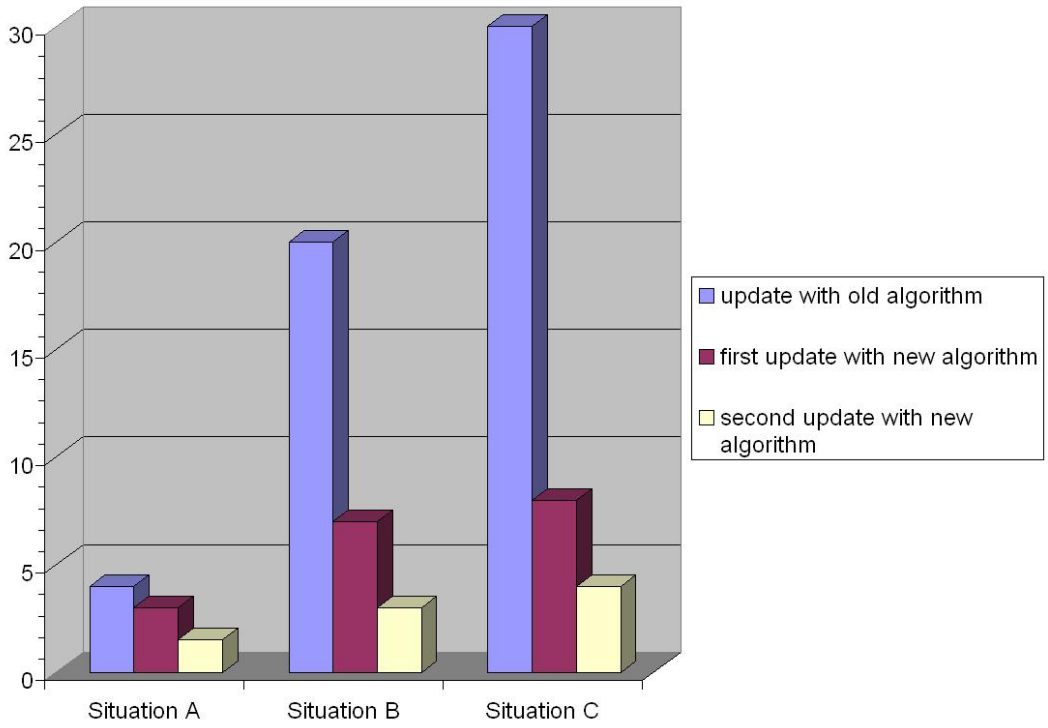


Figure 5.1: Update speed in seconds: The old versus the new Algorithm.

Please note that the figures are only snapshots of the systems performance. Many disturbance variables make exact, reproducible measurements nearly impossible:

- The hosts DWARF runns on, use multiuser operating systems with lots of daemons running in the background. As the response times are also depenent of the computing power of every single host: the number of processes running the performance of the operating system and so on.

- Furthermore, DWARF runs on computers, that are also used for other purposes: office tasks, software development for example. So while making a measurement the load of the hosts can change due to other users and thus intoduce errors.

- The hosts in the DWARF network have different hardware configurations and thus different computing power. So the steps from situation A to sitiation C are not linear[1]. The host atbruegge9 for example is one of the weakest in the network as can

[1]That's why the situations are not numbered but tagged with letters.

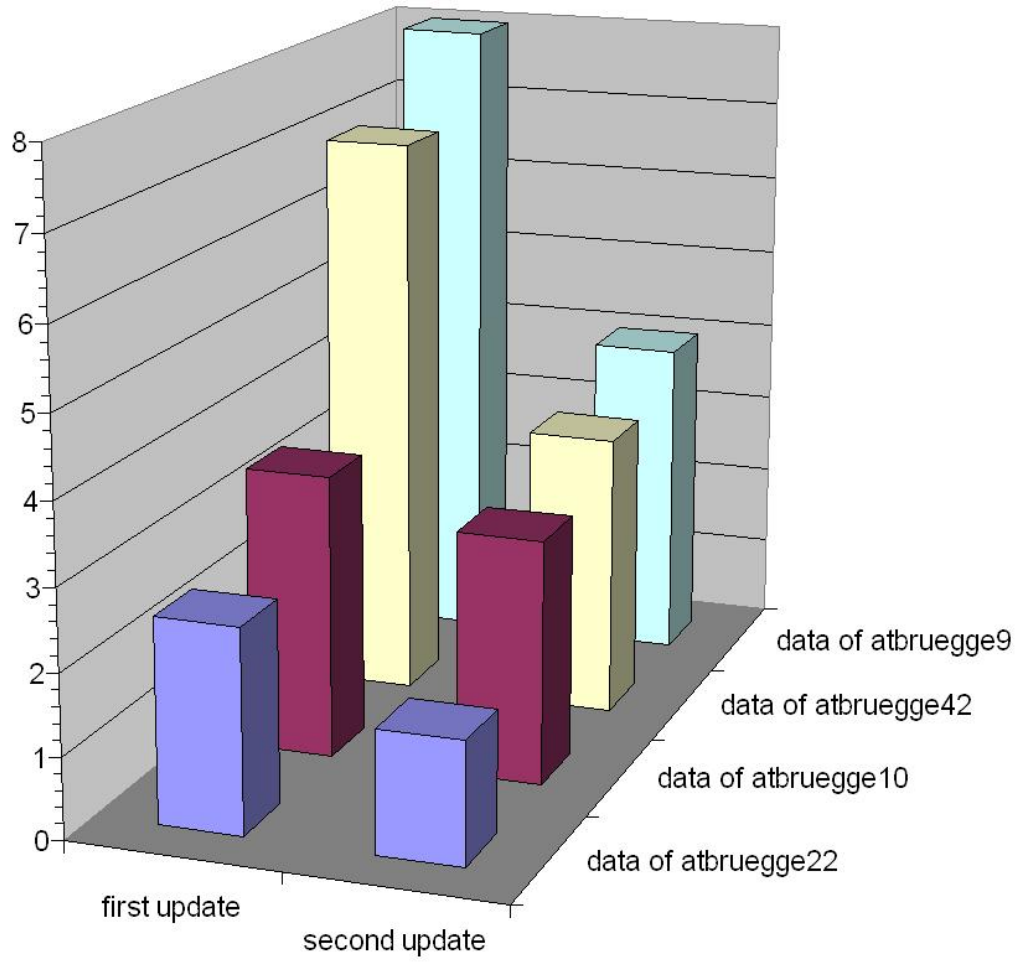


Figure 5.2: Update speed in seconds in situation C: This diagram shows how long it takes for the data of the different hosts to arrive at DIVE.

be clearly seen in Figure 5.2: Its response takes much longer than the ones of the other hosts.

However the measurements clearly show the impact of the new update routine: It is significantly faster than the old one.

5.2 Feedback

Before trying to give ideas and advice for future work, let us look back: what lessons have been learned? More specific: what extensions or changes constituted in the requirements list were valuable, useful or accepted by the users? As at the very end it is always the end-user who decides about failure or success of a piece of software, it makes sense to ask him. Here is a list of the main features developed during this SEP:

- Changing Predicates
- Changing Attributes
- Connecting Services
- Starting Services
- Configuring Services
- Saving the XML Description
- Higher Update Speed
- Color Scheme for Services
- List View
- Grouping of Services

The team members of CAR were asked to rate the value of each of them in the range of 1 to 5, with 5 being the best mark. The CAR-Team consisted of students of different experience levels: three wrote their diploma thesis and four made a SEP[2]. These two different groups are nearly similar to the two user types identified in the requirements analysis:

1. Component Developer (three Diploma thesis writers concentrating on specific parts of the system)
2. Application Developer (four SEP-students concentrating on authoring)

[2]System-Entwicklungs-Projekt

So the feedback is divided in these two parts. In addition to the rating of features I discussed some aspects with CAR team members because only anonymous grades do not tell the whole story.

Figure 5.3 shows the results of the ranking. I will now summarize some opinions that where predominant in the discussions. Concerning the list view some SEP team members told me that as update speed increased, they did not use or did not need it anymore. Another point announced by some of the diploma thesis writers was that the list view would be very useful but the tree structure of it collapsed after every update and thus rendered this view more or less useless for their work. This problem is due to the simplistic update routine of this dialog and could be an issue for future work.

The grouping feature for services was considered more or less lightweight for their work by both groups. However, in the next section we will see that this feature is in fact very useful for a special scenario.

By some team members I was told that the value of all features is rather minor compared to the value of two improvements: Update speed and stability. In fact I was told that without these they would not have used DIVE at all.

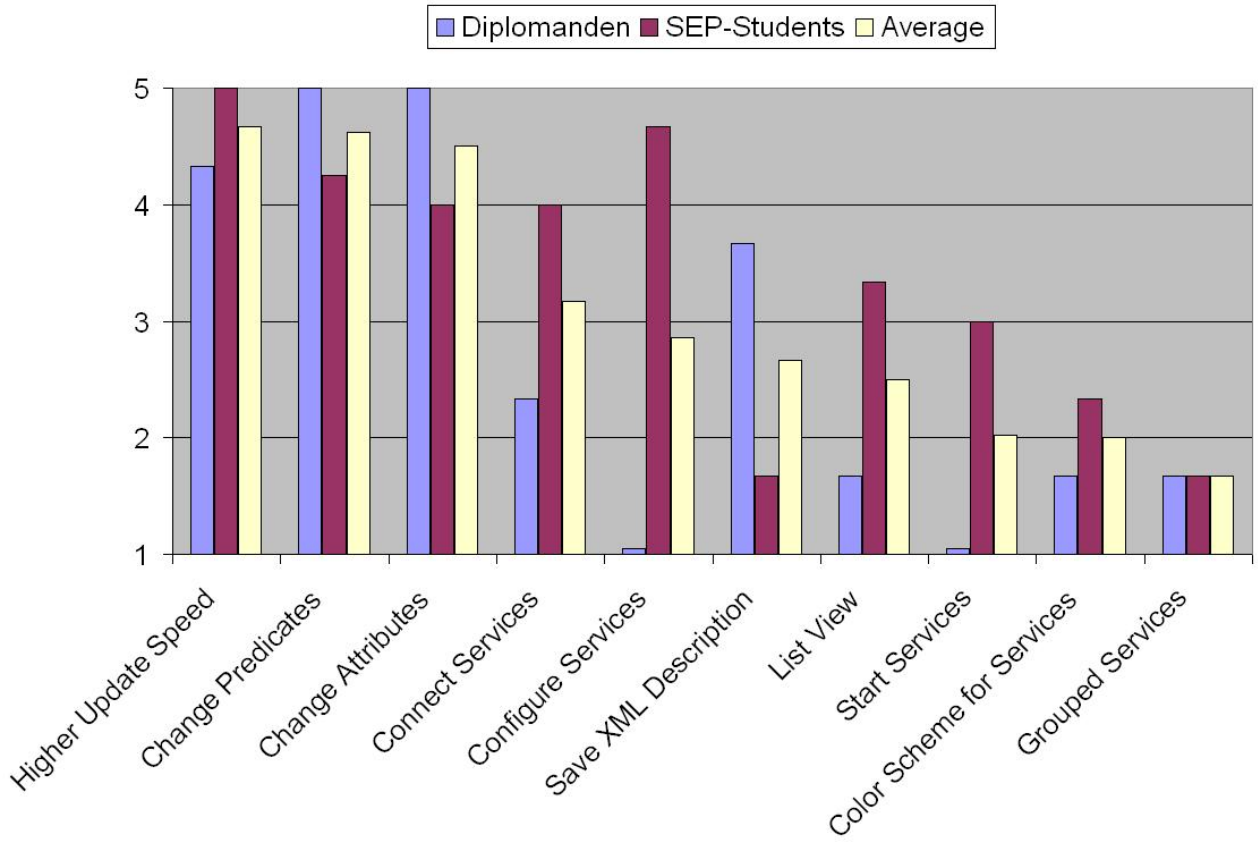


Figure 5.3: The results of the survey.

5.3 Discussion

here are some interpretations of the feedback that may guide the development in the future:

component developers vs. application developers It can be clearly seen in the Figure, that component developers are mainly interested in changing attributes and predicates. This is quite understandable as their main concern is the fine tuning of only one or two services. However, application developers are interested in a great variety of features. This picture is congruent with the assumptions made in the use case specification.

the list view As I already mentioned, the value of the list view could be heavily improved by an iterative update routine, that does not collapse the tree list after every update. However, one should ask oneself whether it is worth the effort because due to higher update speeds at least for the SEP-students the demand for a list view seemed to be of decreasing importance.

starting services Starting services did not have the impact I expected them to have. There are most probably two reasons for the non-acceptance of this extension: First, this feature seems to contradict the DWARF principle. If a service is needed and the auto-start ability is set, the service starts anyway. And if the auto-start ability is not set, nobody seems to need it to be started. Personally I favour the second explanation: During the discussions I got to know that also the auto-start ability is not really used or better to say not accepted by the DWARF developers. So, an expansion that is based on it is likely to share the same lot. But why is the auto-start ability not used? Maybe the system becomes too dynamic for humans to control? Surely, answering this question would be an interesting piece of psychological research.

grouping of services and color scheme According to the survey the color scheme and the grouping function seem to be quite useless. But this is not the case. We omitted one class of actors: the visitor. The two features turned out to be particularly useful for presenting the system architecture to visitors; the first time during the final CAR-presentation. This experience matches the planning, documented in the use cases, especially the use case `Demonstrate System Design`.

effort versus effect One of the most demanding features to implement was surely the new update routine. But as the survey clearly shows is was also the most valuable. The features that were very usefull and at the same time easy to implement were the "change predicates" and "change attributes" features. Finally a feature of less relevance but at the same time even less effort to implement: the color scheme.

speed and stability As already mentioned, increasing update speed and increasing stability turned out to be the major keys to attain user acceptation. And this is surely also true for the future. In my opinion, making DIVE more user friendly and convenient is the predominant task of every future work on DIVE.

Chapter 6

Future Work

> Vorhersagen sind sehr schwierig, insbesondere, wenn sie die Zukunft betreffen[1].
> *Karl Valentin*

6.1 Near Future

Now, let us have a look at features that might be incorporated in the next generation of DIVE. We will first look at some refactoring measures and than discuss extensions.

6.1.1 Refactoring of DwarfSystemModel

The DwarfSystemModel is derived from the abstract class DWARFModelEnumerator. Which should define an interface according to the Iterator/Enumerator Pattern known from JAVA Collections [20], the STL [18] or Algorithm Design in general [14]. But there has been a problem: The class DwarfSystemModel is used by many different Threads. Now the Pattern has been corrupted to face this problem: The method to iterate one step further was supplied with an argument denoting the current position. This is a design fault because it destroys the initial benefit of the pattern: freeing the programmer of keeping track of the current position. The DwarfSystemModel should never be an Iterator itself but produce Iterators, one for each Thread. For the retrieval of Service groups, this has already been done and the added argument is not used anymore. Now it is time to refactor also the DwarfSystemModel for the retrieval of single Services and to eliminate the added argument.

6.1.2 Multible Model Views

In the initial DIVE there has been only one View on the Model: the GraphView. This has changed. Now a ListView was added and there will hopefully come more. The problem is that the Model-View-Controller Pattern that was used was not implemented correctly: It

[1] Forecasts are very difficult, especially when they pertain to the future.

was only implemented to deal with one View, the GraphView. According to the "pattern litriture" [3, 7, 2] the MVC should be implemented to deal with many Views on one Model. Hence there should be an interface for the View that is implemented by the DWARFGraphView and the ListView. The Views could then be held in a convenient way by the Application.

6.1.3 Cleaning up the DIVEApplication Class

Although the class `DIVEApp` has been freed of dislocated code there has to be done more. Still some dialogs are misusing `DIVEApp` to host their update procedures. The aim should be to decrease `DIVEApp's` size drastically by outsourcing this kind of code. Also `DIVEApp` is subject to changes that are induced by the suggestions in the previous section "Thoughts about the Model Views".

6.1.4 Polishing up the GUI

Despite my efforts to make the UI more comfortable the use, it remains a typical "Computer Scientist's UI". A more fundamental approach than just beautifying some Dialogs is necessary to make the GUI really ergonomic. Maybe an expert on User Interfaces should be counselled. In my opinion a general design like Microsoft uses for its Visual Studio [15] or IBM for its Eclipse [11] would already be a big step in the right direction: The ListView and the PropertiesDialog should not float around but be integrated in the main window area as Panels. Figure 6.1 shows the typical eclipse layout with panels surrounding the main window area.

As Eclipse is a very extandible development environment, an integration of DIVE into Eclipse is worth thinking of. The drawback: DIVE had to be completely reimplemented because Eclipse requires extentions to be written in Sun Microsystem's Java [20].

Another quite interesting point would be to adapt the graph layout to the UML-Standard [2, 5]. However this would require major changes in the GraphView class.

6.2 Far Future

6.2.1 DIVE in 3D?

The nets in DWARF can become rather vast and the two dimensions of the screen soon lead to a confusing graph. So the idea of a three-dimensional view on the network is quite obvious. Maybe in combination with some stereo vision technique. Already Daniel Pustka presented the idea of a 3-dimensional graph in his work [16]. Of course also the difficulties are quite obvious: How to layout this graph? And, will a three-dimensional graph be really more comprehensible?

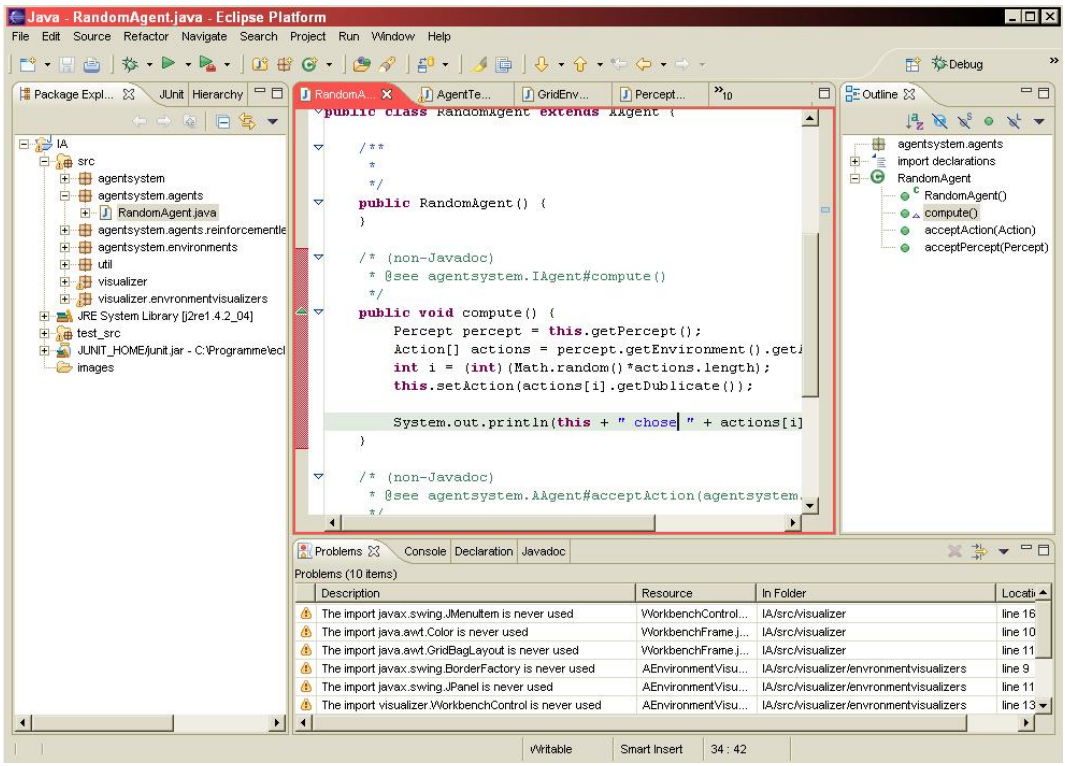


Figure 6.1: The typical eclipse layout with panels surrounding the main window area.

6.2.2 DIVE in AR?

Another idea, that even goes one step further, is the idea of making DIVE an augmented reality applicationitself. It has also been discussed in [17] in the context of dynamic labelling. The Services could be presented as augmentation in the real world. A tracking service would, for example, be floating near the tracking cameras. But also in this approach the difficulties are quite obvious: Where do services go that do not have a direct connection to the real world, like filters? How can DIVE help to debug and build an AR-System if it is built on this very System and does not run on an simple workstation?

Bibliography

[1] M. Bauer, B. Bruegge, G. Klinker, A. MacWilliams, T. Reicher, S. Riss, C. Sandor, and M. Wagner, *Design of a Component–Based Augmented Reality Framework*, in Proceedings of the International Symposium on Augmented Reality – ISAR 2001, New York, USA, 2001.

[2] B. Bruegge and A. H. Dutoit, *Object-Oriented Software Engineering: Conquering Complex and Changing Systems*, Prentice Hall, Upper Saddle River, NJ, 2000.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerfeld, and M. Stal, *A System of Patterns, Pattern-Oriented Software Architecture*, Wiley, West Sussex, England, 5th ed., 2000.

[4] Chair of Applied Software Engeneering, *The DWARF Wiki-Web*. `http://www.augmentedreality.de`, March 2004.

[5] T. Erler, *UML*, Verlag Moderne Industrie Buch AG and KG, Landsberg, 2001.

[6] F. S. Foundation, *GNU General Public License*. `http://www.gnu.org/copyleft/gpl.html`, June 1991. version 2.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*, Addison Wesley Pub Co, 1995.

[8] O. M. Group, *The Common Object Request Broker: Architecture and Specification*. `http://www.omg.org/technology/documents/vault.htm#CORBA_IIOP`, July 1999. CORBA 2.3 Specification.

[9] P. Hallama, M. Schwinger, S. Korbinian, and N. Doerfler, *Entwicklung eines dynamisch ueber Tools modifizierbaren Filternetzwerkes*. `//???`, April 2004.

[10] A. Hunt and T. David, *The Pragmatic Programmer*, Addison Wesley Professional, 1999.

[11] IBM, *IBM's Eclipse Homepage*. `http://www.eclipse.org/`, March 2004.

[12] A. MacWilliams, *DWARF – Using Ad-Hoc Services for Mobile Augmented Reality Systems*, Master's thesis, Technische Universität München, Department of Computer Science, Feb. 2000.

[13] A. MacWilliams, C. Sandor, M. Wagner, M. Bauer, G. Klinker, and B. Bruegge, *Herding Sheep: Live System Development for Distributed Augmented Reality*, in To appear in Proceedings of the International Symposium on Mixed and Augmented Reality (ISMAR), Tokyo, Japan, 2003.

[14] G. Michael and T. Roberto, *Algorithm Design*, Wiley, Hoboken, USA, international edition ed., 2002.

[15] Microsoft, *Microsoft's Visual Studio Homepage*. `http://msdn.microsoft.com/vstudio`, March 2004.

[16] D. Pustka, *Visualizing Distributed Systems of Dynamically Cooperating Services*. `//wwwbruegge.in.tum.de/pub/DWARF/OberSeminar/SEPPustka.pdf`, March 2003.

[17] Riedl, *Automatic Layout Of User Interface Widgets In Augmented Reality*. `//???`, Mai 2004.

[18] Silicon Graphics, *Silicon Graphics' STL Homepage*. `http://www.sgi.com/tech/stl/`, March 2004.

[19] B. Stroustrup, *Die C++-Programmiersprache*, Addison-Wesley-Longman, Bonn, 3rd ed., 1998.

[20] Sun Microsystems, *Sun's Java Homepage*. `http://java.sun.com`, March 2004.

[21] Trolltech, *Trolltech's QT Homepage*. `http://www.trolltech.com/developer/`, March 2004.

[22] CAR *Project Homepage*. Technische Universität München, `http://wwwbruegge.in.tum.de/projects/lehrstuhl/twiki/bin/view/DWARF/ProjectBar`.

[23] van Heesch, Dimitri, *The Doxygen Homepage*. `http://www.doxygen.org/`, March 2004.