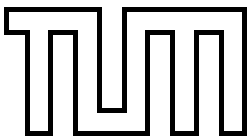
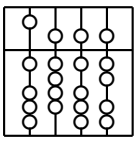


Technische Universität München
Fakultät für Informatik



Systementwicklungsprojekt

Visualizing Distributed Systems of Dynamically Cooperating Services

Daniel Pustka

Aufgabensteller:	Prof. Dr. Bernd Brügge
Betreuer:	Dipl.-Inform. Asa MacWilliams
Abgabedatum:	15. März 2003

Erklärung

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

München, den 10. April 2003 Daniel Pustka

Contents

Chapter 1

Introduction

To facilitate the development of distributed augmented reality applications, the chair of applied software engineering at TU München has developed the DWARF software architecture. The heart of this architecture is a middleware which dynamically locates and connects services distributed across a network.

Applications built on top of DWARF inherently are self-assembling. Developers who are working with DWARF systems often face the problem that it is difficult to see how the components, spread all over the network, are connected by the middleware.

The solution to this problem proposed here is DIVE (DWARF Interactive Visualization Environment), a visualization tool for DWARF systems which I have designed and implemented as a system development project (SEP).

This document is the DIVE manual both for users and developers. It's basic structure follows the methodology described in [10]. The document starts with a chapter about requirements analysis, followed by a discussion of existing graph layout tools. Chapters 4 and 5 will give details about the system and object design of DIVE. Chapter 6 describes possible future extensions.

New users of DIVE are recommended to read the requirements analysis chapter which shows how the software is operated from a user's perspective. The other chapters are more technical and aimed towards future developers who need to understand the internal structure of DIVE.

1.1 DWARF Overview

Before I can go on explaining DIVE, it is necessary to understand the basic concepts of DWARF. The following section gives a brief overview of the aspects relevant to DIVE. For further reading, an in-depth discussion of DWARF can be found in [13, 8, 1].

A DWARF system consists of several cooperating services, located on different stationary or mobile computers. Each service has a set of needs and abilities, which describe how the service can communicate with other services. The most important property of a need or ability is the protocol that is used for communication. In addition, an arbitrary number

of attributes can be defined which give more detailed information about what kind of data an ability provides or a need expects. To limit the selection of communication partners, predicates can be given, which the other service has to fulfill.

When the supported communication protocols of a need and an ability match and all predicates are satisfied, the DWARF middleware dynamically connects the two services. DWARF is designed for distributed applications, so this works across the network, but preference is given to local services. When new services are added to the system or existing ones are removed from it, the middleware reacts to these changes and adds or removes connections between services.

The part of the DWARF middleware which is responsible for locating and connecting services is called the service manager. Usually one instance is running on each computer, acting on behalf of the local services and advertising them on the network.

1.2 Problem Statement

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.

The purpose of this SEP is to develop a visualization tool for DWARF systems. The software must take account of the distributed aspect of DWARF systems by collecting information about the services from the different service managers. This information is then used to draw a two-dimensional diagram showing the currently active services and the connections between them. Caring about the dynamic aspect of the system, this diagram should always be kept up-to-date to reflect the changes in the system.

A diagram can only contain limited information about the services themselves. Therefore, by clicking on a component in the diagram, a developer should be able to get additional information about it, such as service attributes, state of the service, name of the host computer, communication protocols, etc.

Another problem comes up when the system is not working as expected. If the reason is not immediately clear, the communication between the services should be among the first things that are checked. To help in the debugging process, the visualization tool should provide means to monitor the communication between services. The first version only includes support for CORBA structured events, but the application should be extensible to other forms of communication.

Chapter 2

Requirements Analysis

This chapter gives a more formal and more detailed description of the requirements for a DWARF visualization tool. The chapter's structure roughly follows the guidelines for Requirements Analysis Documents (RADs) given in [10].

Readers who are new to DIVE, should read the section about functional requirements in order to get an overview of the tool. If you want to learn how to use DIVE, reading the use cases is recommended as they show step-by-step how to operate the application.

2.1 Functional Requirements

The following sections give a high-level overview of the functionality of the DIVE tool.

Collect Structural Information The information about the structure of a DWARF system usually is distributed across several computers. Each service manager only knows it's own services. The visualization tool therefore first has to discover all running service managers. It then contacts each of them in order to get information about the state of it's services and the connections between them. This data has to be combined to form a complete and consistent picture of the system.

Display Graph and Attributes The main screen of the visualization tool displays a two-dimensional graph showing the services as nodes. The edges represent the connections between services that have been established by the service manager.

Services are shown in a fashion similar to UML class diagrams. The service name is on top with the service's needs and abilities listed below. A preference dialog, available from the menu, allows the user to switch off the needs and abilities, giving a more compact view of the system.

Additional attributes of a service, a need or an ability can be obtained by clicking on it. Such attributes could be service state, hostname or supported communication protocols.

The user can choose to refresh the graph manually or let the application do that in fixed intervals.

Information Filtering To reduce the complexity of the graph, the visualization tool provides settings that allow the user to view only parts of the DWARF system. The user should be able to select or hide services that have certain attributes.

Export To File The application allows the graph to be exported to a PostScript file that can be imported into a graphics or word processing application.

Debugging When the user has selected a need or an ability that supports the "CORBA event sender" communication protocol, a button labeled "View Events" is available. When clicking on it, a new window opens which displays all CORBA events that are sent by the selected need or ability. If the event contains composite data structures, they are decomposed into their elements to make the content human-readable.

2.2 Nonfunctional Requirements

> "Nonfunctional requirements describe user-visible aspects of the system that are not directly related with the functional behavior of the system." [10]

Graphical User Interface The user interface should be intuitive to use and provide a familiar "look and feel".

User Preferences All preference settings are automatically stored in a configuration file without requiring further interactions by the user.

Response Time All tasks that may possibly last longer than two seconds shall be performed in the background without interrupting the user's work.

Extensibility DIVE should be extensible, especially with respect to new user interfaces and debugging of other communication protocols. More extensions are proposed in chapter 6.

2.3 Pseudo Requirements

> "Pseudo requirements are requirements imposed by the client that restrict the implementation of the system." [10]

The DWARF Software Environment

The visualization tool must compile and run in the current DWARF software environment. Therefore it must be compatible with the following software:

- Linux 2.4
- GNU autotools (automake, autoconf)
- OmniORB

2.4 Use Case Models

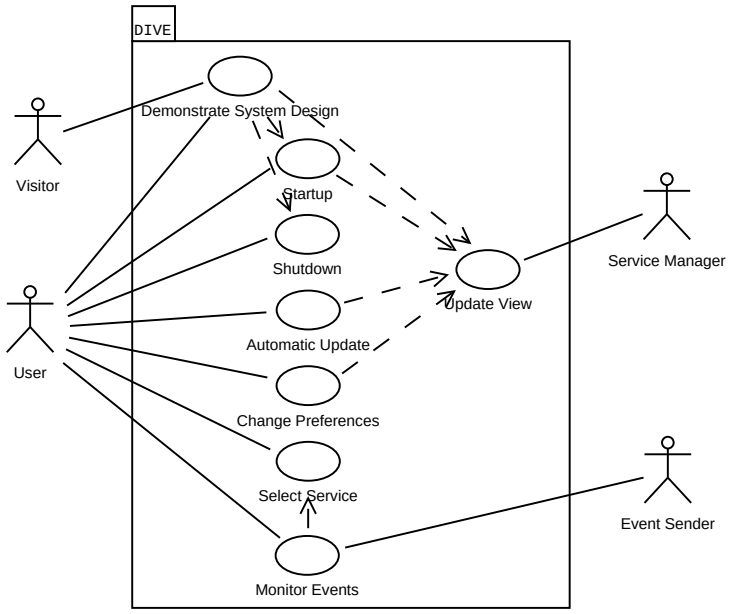


Figure 2.1: Overview of all use cases. The dashed lines represent an <<includes>> relationship.

2.4.1 Actors

User The `User` is the main user of the DIVE tool described in this document. He uses DIVE for debugging and to explain the DWARF system to `Visitor`s.

Visitor The `Visitor` does not directly interact with DIVE. Instead, the `User` uses DIVE to explain a DWARF system to a `Visitor`.

DWARFServiceManager The `DWARFServiceManager` is the most important part of the DWARF middleware. An overview of how it works was given in section 1.1. DIVE communicates with the `DWARFServiceManager` in order to retrieve information about the DWARF system.

EventSenderService The `EventSenderService` is a DWARF service which has a need or ability of type PushSupplier, which sends asynchronous events to other services. The use cases show how DIVE can be used to read those messages.

2.4.2 Use Cases

Demonstrate System Design

This use case describes how DIVE is used by a `User` in order to demonstrate a system design to a `Visitor`.

Participating actors	Initiated by `User` `Visitor`
Entry condition	1. The `User` starts the application (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. The `User` continues with steps 2 and 3.
Exit condition	5. The `User` closes the application (includes `ProgramShutdown`).

Monitor Events

This use case shows how DIVE is used to monitor events from an `EventSenderService`.

Participating actors	Initiated by `User`
	`EventSenderService`
	`DWARFServiceManager`

Entry condition	1. The `User` selects an ability of type PushSupplier (includes `SelectService`).
Flow of events	2. The `User` selects "View Events" in the ability's properties window. 3. DIVE opens a new event monitoring window. 4. DIVE creates a new `EventReceiverService`. 5. The `DWARFServiceManager` connects the newly created service with the `EventSenderService` selected in step 1. 6. The `EventSenderService` sends an event which is received by the `EventReceiverService`. DIVE parses and displays it in the `EventMonitorWindow`.
Exit condition	7. The `User` closes the `EventMonitorWindow`.

Program Startup

This use case describes the actions performed by DIVE at program startup.

Participating actors	Initiated by `User` `DWARFServiceManager`
Entry condition	1. The `User` starts the application.
Flow of events	2. The application displays the main application window. 3. The application loads the `User`'s preferences from a configuration file. 4. The application establishes a communication channel with all available `DWARFServiceManagers`. 5. DIVE gathers information about running services from the `DWARFServiceManager`s and presents it as a graph in the `GraphView` (includes `UpdateView`).
Exit condition	6. The application is ready respond to further commands from the `User`.

Program Shutdown

The Program Shutdown use case describes what happens when the application is closed.

Participating actors	Initiated by `User`
Entry condition	1. The `User` chooses "Quit" from the menu.
Flow of events	2. DIVE removes all application windows. 3. The application stores the user's preferences into the configuration file. 4. DIVE cuts all communication channels to `DWARFServiceManager`s and `EventReceiverService`s.
Exit condition	5. The application is completely removed from memory.

Update View

Update View describes how the application retrieves the system structure and presents it to the user.

Participating actors	`User` `DWARFServiceManager`

Entry condition

1. The Update View use case is invoked in three different ways:
   - Automatically at program startup.
   - Automatically by a timer when "Automatic Updating" is enabled.
   - Manually when the `User` selects "Update View" from the menu.

Flow of events

2. The application contacts all known `DWARFServiceManager`s and gets their information about the DWARF system. This information is used to update the `DWARFSystemModel`.
3. DIVE filters the information and creates a graph description which is passed to the `GraphLayouter`.
4. The application displays the resulting graph arrangement in the `GraphView`.

Exit condition

5. The `GraphView` shows the current state of the system.

Special requirements As steps 2 and 3 might consume an arbitrary amount of time, they are performed in the background without interrupting the user's work.

Change Preferences

In this use case I describe, how the `User` can change the application's settings. Examples for such settings are: Path to layout program, display size, graph details, etc.

Participating actors Initiated by `User`

Entry condition

1. The `User` selects "Preferences" in the "View" menu.

Flow of events

2. The application displays a dialog which contains controls for all settings.
3. The `User` changes the settings and confirms by clicking the "OK" button.
4. The application updates the display with the new preferences (includes `UpdateView`).

Exit condition

5. The display conforms to the new preferences.

Automatic Updating

This use case describes how the `User` tells the application to update the graph in fixed intervals.

Participating actors	Initiated by `User`
Entry condition	1. The application has started and "Continuous Updates" is disabled.
Flow of events	2. The `User` selects "Continuous Updates" in the "View" menu. 3. The "Continuous Updates" menu item gets a check mark to indicate that the function is activated.
Exit condition	4. The graph display is updated every 5 seconds (includes `Update View`).
Special requirements	Automatic updating can be disabled in the same fashion.

Select Service

This use case describes how a user gets additional information about a service.

Participating actors	Initiated by `User`
Entry condition	1. The application has started and some services are visible in the main window.
Flow of events	2. The user clicks on the name of a service. 3. All connections of the selected service are highlighted.
Exit condition	4. A new `PropertiesWindow` becomes visible. It shows a list of all attributes of the selected service.
Special requirements	There is always at most one `PropertiesWindow` visible. If there already is one, it will be reused for the new service.

2.5 Object Model

This section introduces the analysis objects that can be extracted from the use cases.

DWARFSystemModel

The `DWARFSystemModel` contains the application's knowledge of the DWARF system structure. The model not only stores that information, but also is responsible for acquiring it and keeping it up-to-date. The `DWARFSystemModel` registers itself as a service within the DWARF middleware in order to get connections to all service managers. Therefore the `DWARFSystemModel` is both an entity object as well as a boundary to the DWARF system.

The `DWARFSystemModel` has information about services, their needs and abilities and about sessions. Sessions represent the connections between services.

UserPreferences

The `UserPreferences` are all settings that the `User` can change in the application. This includes the options that can be explicitly specified in the configuration dialogs, but also implicit parameters like window sizes and positions. For commodity the preferences are automatically stored in a configuration file and loaded at program startup.

GraphView

The `GraphView` is the part of the user interface which displays information as a graph. It should be able to perform automatic graph layout.

PropertiesWindow

The `PropertiesWindow` is a window that is used to display additional information about a service, a need or an ability. It contains a list with all attributes that are made available by the `DWARFServiceManager`. A `PropertiesWindow` pops up when the user clicks on an object in the `GraphView`.

EventReceiverService

The `EventReceiverService` is a boundary object to other DWARF services. The application creates an instance of this service when the user wants to view the events that are sent by a PushSupplier ability. The service has it's needs setup such that the service manager always connects it to the selected PushSupplier ability.

EventMonitorWindow

The `EventMonitorWindow` displays the events that are received by an `EventReceiverService`. The events are decomposed into it's elements and displayed as a tree with two columns. The left column shows the name of the element, the right one it's value.

PreferencesDialog

The `PreferencesDialog` can be activated from the menu. It contains controls for all global application settings.

2.6 User Interface

The following pictures give an overview of what DIVE looks like.

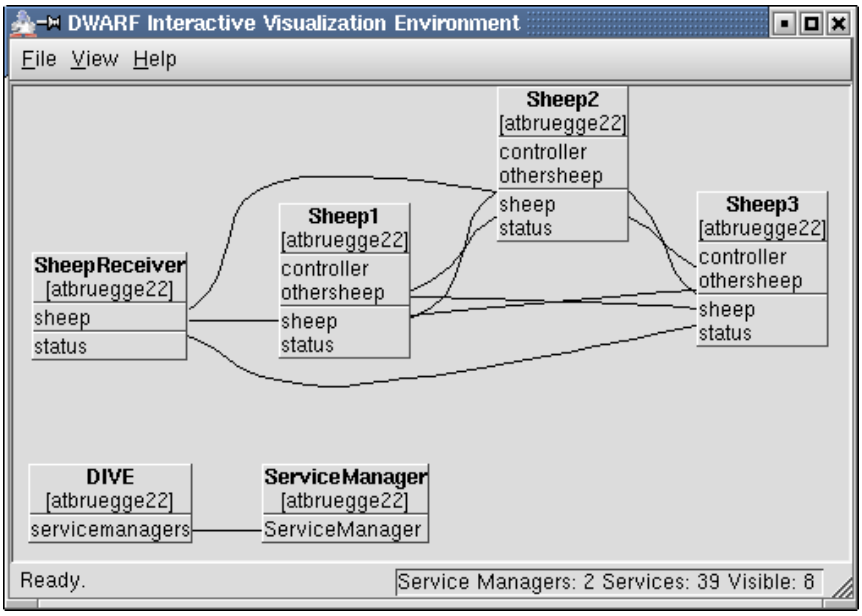


Figure 2.2: DIVE's main screen. It shows how three Sheep services are connected to each other and to a SheepReceiver. We also see how DIVE is connected to a ServiceManager.

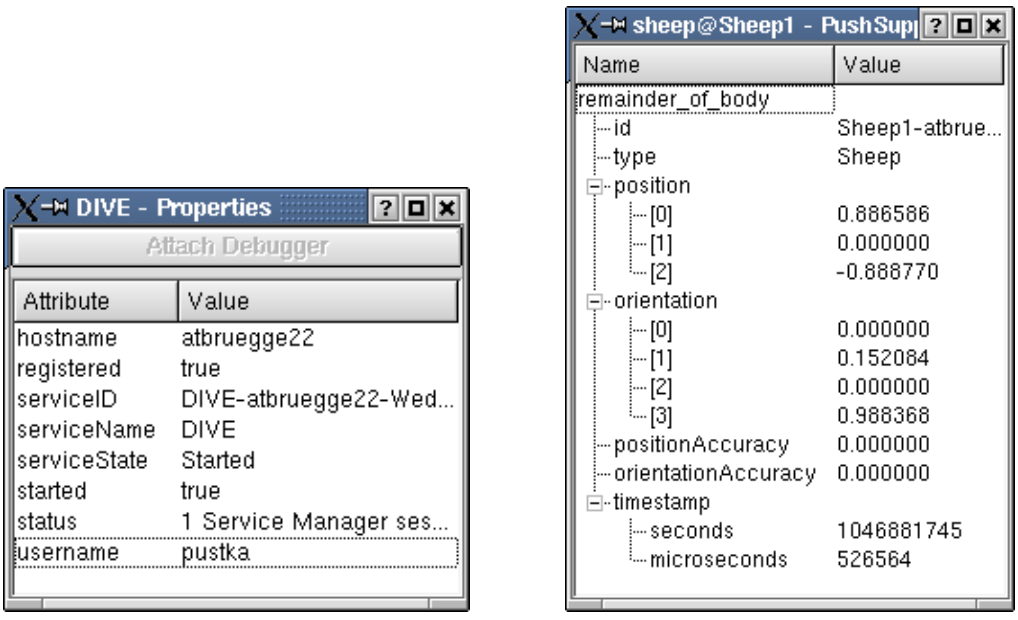


Figure 2.3: On the left hand side we see a `PropertiesWindw` displaying the attributes of the DIVE service. The right image shows an `EventMonitorWindow` receiving PoseData from a Sheep service.

Chapter 3

Existing Tools

In the following chapter I will justify my choice of a graph layout tool. It may also serve as a starting point for anyone who wants to make an application that displays graphs with automatic layout.

3.1 Overview of Graph Visualization and Layout

To be able to display the system as a graph, there first has to be a way to arrange the nodes and edges automatically. As graph layout still is an active research area, doing it on our own go would far beyond the scope of this project. Fortunately, many programs and libraries for that purpose are available which will be discussed in this section.

There are basically four different ways to build an application with graph layout capabilities:

1. Use an existing graph viewing tool with it's own GUI and extend it. This can be relatively easy if the tool has an API or an interface to scripting languages.

2. Write your own GUI but use an existing graph editor component. The application writes the graph into a data structure and passes it to the graph component which does layout and rendering. User input is handled through callback functions.

3. Write your own GUI and use a library for graph layout only. The application writes the graph into a data structure which is passed to the layout library. Drawing of the resulting arrangement is done again by the application.

4. Write your own GUI and call an external program for layout. This is basically the same as using a library, but communication is not done through data structures in memory but by exchanging textual graph descriptions through Unix pipes or temporary files. Additional overhead is introduced by string parsing and process creation.

There are a number of features that are common among many graph layout tools. The following list describes those that might be useful for DIVE.

Incremental Layout Takes the description of a graph where some nodes already have a position. In the resulting arrangement these nodes should change their position only when absolutely necessary. Incremental layout is particularly useful for visualization of dynamic systems. When there is only a small change in the system, the changes to the graph arrangement should also be small.

Records To allow the needs and abilities to be displayed in a UML-like fashion, the graph layouter should allow nodes to be divided into multiple fields (records). Routing edges from one record to another should also be supported.

Ports This concept allows the user to specify areas on the border of a node where edges can be placed. Ports allow the creation of effects similar to records.

Clusters or subgraphs To increase the clearness of the diagram, it should be possible to arrange nodes with similar properties next to each other. This could be useful to group together DWARF services which run on the same computer or belong to the same subsystem.

To allow the visualization tool to be distributed together with the DWARF middleware, the following discussion will focus on software which is available under a open source license. Commercial programs or libraries are only mentioned for completeness.

3.2 Existing Graph Visualization and Layout Tools

This section gives an overview of exiting graph layout tool that are available on the internet.

VCG Tool [22]

Author	Universität des Saarlandes, Saarbrücken
License	open source (GPL)
Distribution	source and binary

The VCG (Visualization of Compiler Graphs) tool is one of the two most popular freely available graph layout tools. Unfortunately it has not been updated since 1995 because the developer now works for ILOG ("JViews", see below) and was employed by TomSawyer Software before ("Graph Layout Toolkit", see below).

The software reads a textual graph description and outputs the graph as a PostScript or bitmap image or displays it in a X11 window. Unfortunately there is no output format that only includes the coordinates of nodes and edges, making it practically impossible to use only the layout capabilities without actually drawing a diagram. VCG supports both records and incremental layout.

The C source code of the whole software is available. However the part with the actual layout algorithms is "uglified" so there is no practical way to modify it. Extraction of the relevant parts still is possible, but would be time-consuming due to the lack of documentation.

Graphviz [5]

Author	AT&T Labs Research
License	open source
Distribution	source and binary

Graphviz is a collection of standalone programs for graph layout. The dot layouter uses a hierarchical algorithm that tries to find tree-like structures in the input graph. Neato is a so-called "spring layouter" which uses a system of attracting and repelling forces, trying to minimize the overall energy. The other programs include dotty, a graph viewer for X11 displays.

Both layout programs read the graph description in the proprietary dot format. Available output formats include, among various vector and bitmap graphics types, the "plain" format which is simply a list of coordinates that can easily be parsed for display in other programs.

The graph layout algorithms are located in libraries that could be used directly, but unfortunately the interface was not documented by the time this project started. Dot supports records and clusters. Neato can be used for incremental layout - but does not fully support records and clusters.

Graphlet [20]

Author	Universität Passau
License	free for noncommercial use
Distribution	binary, source code on request

Graphlet is a toolkit for graph editing and drawing applications. It consists of a graph editor which can be extended using a scripting language based on Tcl/Tk. Unfortunately, even the binary distribution is only available upon request.

LEDA [4]

Author	Algorithmic Solutions
License	commercial
Distribution	binary and source code licenses available

LEDA is a library for efficient data types and algorithms. It includes algorithms for graph layout as well as a window component for graph visualization and editing. LEDA now is a commercial product, although previous version developed at MPI were available for free.

VGJ [6]

Author	Auburn University
License	open source (GPL)
Distribution	java source code

VCJ is a graph editor written in Java. It's layout capabilities are not as advanced as those of VCG or Graphviz. VCJ does not support records and is not specifically designed to be used in other applications.

AGD [3]

Author	TU Wien, Uni Köln, MPI
License	free for academic use, registration required
Distribution	binary only (Linux/Sparc/Windows)

AGD is a library of algorithms for graph drawing. It uses the commercial LEDA libraries (see above) and includes a large number of graph layout algorithms.

GDToolkit [21]

Author	Università di Roma Tre
License	commercial, free for academic use (limited version), registration required
Distribution	binary only (Linux/Sparc/Windows)

Library for graph layout, based on the commercial LEDA library. Can be used as C++ library or as a standalone program. Unfortunately only limited versions are available for free.

JViews [12]

Author	ILOG
License	commercial
Distribution	binary (JavaBeans)

Commercial Java packages for graph layout and display.

Graph Layout Toolkit [18]

Author	Tom Sawyer Software
License	commercial
Distribution	binary (Windows/Linux/Sparc/MacOS9)

Commercial library for graph layout. Includes many different graph layout algorithms, some support ports.

aiSee [2]

Author	AbsInt
License	commercial
Distribution	binary

Commercial standalone graph viewer, based on VCG (see above).

daVinci Presenter [9]

Author	b-novative
License	commercial
Distribution	binary

Standalone program for graph drawing and editing. DaVinci has an API that allows it to be used as a visualization component in other programs. Ports are supported.

GEF [11]

Author	(open source project)
License	open source (BSD licence)
Distribution	source code

The Graph Editing Framework is a java library for graph editing. Unfortunately it's automatic layout capabilities are only rudimentary. GEF is used by the argoUml CASE tool.

3.3 Rationale

The final decision is to use dot from the Graphviz collection for graph layout. It is free software and offers many powerful layout features such as records or clusters. The development of it is ongoing, so bugs are likely to be resolved. DIVE calls dot as an external layout tool, using Unix pipes for communication. Therefore no complex API has to be learned. If performance should become an issue, DIVE can be adapted to use the library version of dot, especially now that the documentation is available.

Another advantage of dot is the compatibility with neato, the other layouter from the Graphviz collection. Although it does not fully support all features, neato can be used if a different layout is desired.

Chapter 4

System Design

The purpose of this chapter is to give an overview over the structure of the application. It is mainly of interest for future developers who wish to extend the application.

The structure of the chapter roughly follows the conventions for system design documents (SDDs) as described in [10].

4.1 Design Goals

Reusable components All components of DIVE should work independently of each other, so they can be reused easily in future projects. This also results in a clean design with clear responsibilities.

Extensibility DIVE should be extensible, especially with respect to the extensions proposed in chapter 6.

Compatibility with DWARF The DIVE tool must work together with the middleware and existing DWARF services.

4.2 Subsystem Decomposition

This section describes how the DIVE architecture is devided into four subsystems.

Overview

The architecture of DIVE follows the Model/View/Controller (MVC) design pattern. The `DWARF System Model subsystem` is responsible for acquiring and storing information about the structure of the system, the `Graph Visualization and Layout` subsystem arranges and displays the graph and the `Application` subsystem acts as a controller.

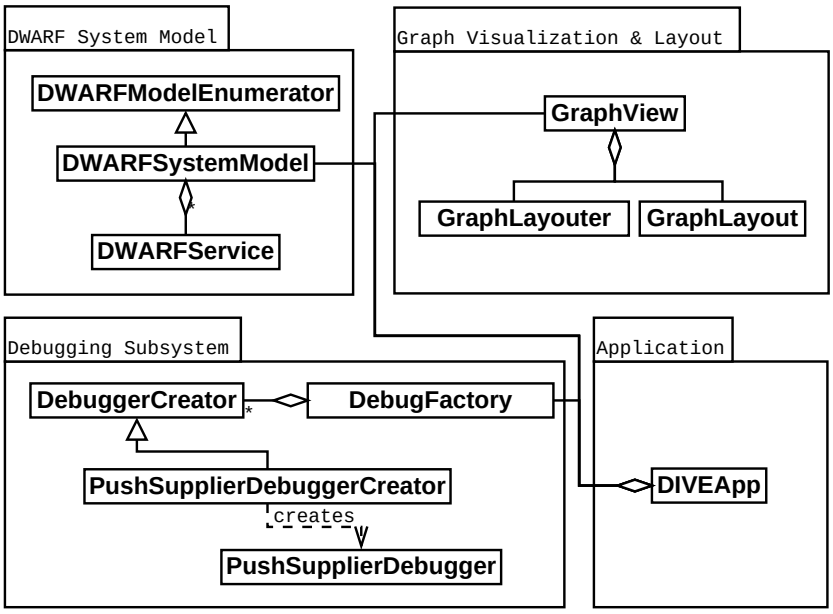


Figure 4.1: The four subsystems of the DIVE architecture with their most important objects.

DWARF System Model

The `DWARF System Model` represents the structure of the DWARF system. It can be seen as the model in the context of a MVC architecture. It's purpose is to connect to the DWARF middleware, to read structural information from the service managers and to make this information available to the rest of the application.

The subsystem runs independently of the other subsystems, so it can easily work with different user interfaces or be reused in other applications. One major design consideration was to provide concurrent access to the data structures, allowing a worker thread to update the data while multiple views are accessing it.

The main classes of the `DWARF System Model` are `DWARFModelEnumerator` which, following the Bridge design pattern, provides an abstract interface to the data structures, and `DWARFSystemModel` which encapsulates the functionality of the whole subsystem in one class.

Graph Visualization and Layout

In the MVC approach, the `Graph Visualization and Layout` subsystem represents the view. It is designed to be a general graph viewer, completely independent of anything DWARF-related. Therefore all user input is delegated back to the application which takes care of the appropriate actions. The subsystem must support records, in order to allow a graph display similar to UML class diagrams.

The most important class is `GraphView`, which encapsulates the functionality of the subsystem in form of Qt widget (see section 4.5). `GraphView` uses the `GraphLayout` class to maintain the graph data structure and `GraphLayouter`, an abstraction for graph layout algorithms (Strategy design pattern).

Application

The `Application` subsystem, in particular the `DIVEApp` class, provides the glue between the components. A `DIVEApp` object is instantiated by the Qt framework. This "root object" is responsible for creating the other objects and for mediating between them. It translates data between the `DWARF System Model` and the `Graph Visualization and Layout` subsystems and reacts to user input. Other features provided by the `Application` subsystem are the main window frame, the menu and the user preference options.

In the MVC pattern, the `Application` subsystem is the controller.

Debugging

The `Debugging` subsystem provides all components that are related to debugging a service, a need or an ability. At the moment only viewing of CORBA structured events sent by a PushSupplier ability is supported, but the subsystem is designed to be extendible to other forms of communication. This extensibility is accomplished by using the factory pattern which is realized in the `DebugFactory` class.

Components of the `Debugging` subsystem should provide their own user interface and have only few interactions with other subsystems.

4.3 Persistent Data Management

The only data that DIVE needs to store persistently are the user preferences. Saving and loading is done using the `Settings` class from the Qt framework which provides simple methods for storing strings and integer values.

4.4 Global Software Control

Event-driven UI As the user interface is based on the Qt framework, it's event-driven architecture is be used, including the Qt-specific signal-and-slot mechanism.

Worker threads DIVE uses worker thread tp perform lengthy tasks which otherwise would disturb the user. Examples for such tasks are updating the `DWARF System Model` and performing the graph layout. Worker threads communicate with the main application by posting asynchronous events into the main event queue, using the Qt event facilities.

4.5 Third-Party Software

As DIVE is a part of the DWARF distribution, third-party-components which are already a fixed part of the DWARF environment (e.g. OmniORB, GNU autotools) are not mentioned here.

DIVE relies on the following additional third party software:

- Qt is used for user interface, global software control and storage of user preferences. As all DWARF applications inherently are multi-threaded, the multi-threaded versions of the Qt libraries (qt-mt) have to be used.
- Graphviz provides the graph layout capabilities (see chapter 3).

Chapter 5

Object Design and Implementation

This chapter is rather technical and intended for future developers who wish to change or extend DIVE. Here I will give general information about how to use the classes that DIVE is made of. If you are interested in the exact signature of each method, you should have a look at the header files. The class and method documentation there is compatible with the Doxygen tool, which can be used to extract a nicely formatted and cross-referenced documentation from the source code.

The chapter is divided according to the four subsystems. Classes that do not belong to a particular subsystem are described in a fifth section.

5.1 DWARF System Model

DWARFModelEnumerator

`DWARFModelEnumerator` provides an abstract interface for concurrent access to the service descriptions stored in the `DWARF System Model` data structures. The methods `GetFirstService` and `GetNextService` are called to enumerate all services. For enumeration string identifiers are used as iterators. The advantage of string identifiers over direct pointers is that enumeration can continue even when the current service was deleted by another thread. This is because enumeration is done in alphabetical order, so the following object can always be determined unambiguously. String identifiers can be narrowed down to a `DWARFService` object by calling `LockService`. The caller must unlock any locked service description after usage by calling the `Unlock` method of the returned object.

DWARFSystemModel

The `DWARFSystemModel` class encapsulates the functionality of the whole `DWARF System Model` subsystem. It is the only object that has to be instantiated by anyone who wants to use the subsystem. The class implements the `DWARFSystemModel` interface for service enumeration.

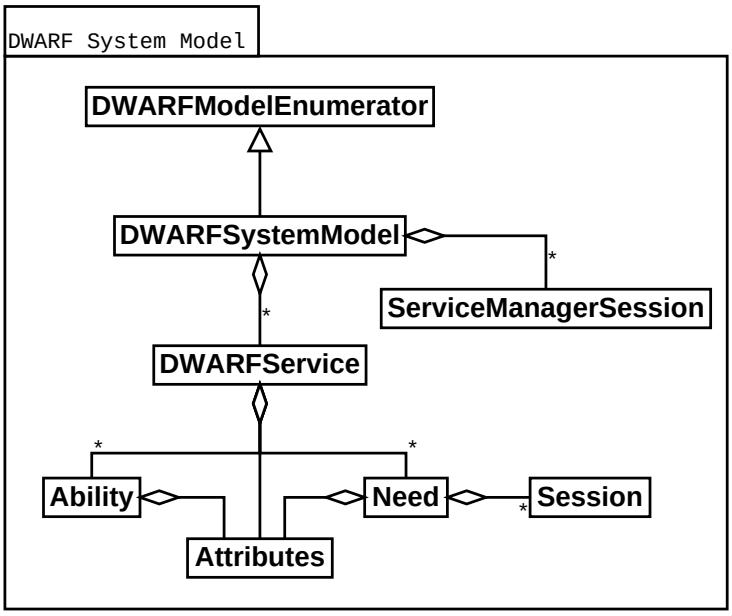


Figure 5.1: Objects of the `DWARF System Model` subsystem.

Before an object of this class is created, the static method `Init` must be called to initialize the CORBA orb. `Init` should only be called once.

In order to find all running service managers, a `DWARFSystemModel` object registers itself as a "DIVE" service within the DWARF middleware. It has a need of type "ServiceManager", which causes the middleware to connect it to all available service managers as the service managers are DWARF services themselves.

ServiceManagerSession

The `ServiceManagerSession` class is used internally by the `DWARFSystemModel`. An instance is created each time the middleware calls the `newSession` method in order to establish a connection to a service manager.

DWARFService

The `DWARFSystemModel` creates an instance of the `DWARFService` class for each DWARF service that is found on the network. The constructor reads all kinds of relevant information from the service. This data is made available by the various `get...` methods. `getNeeds` and `getAbilities` can be called to recursively query the needs and abilities of the service. In order to function in a multi-threaded environment, the class provides `Lock` and `Unlock` methods.

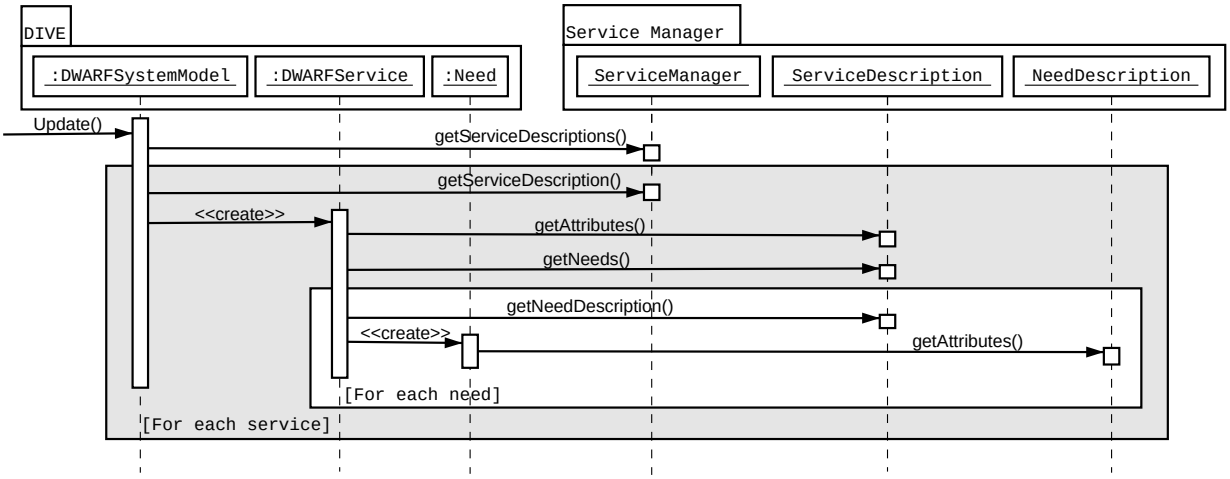


Figure 5.2: Sequence diagram showing the interactions between the classes of the `DWARF System Model` subsystem and the service manager when an update is performed.

Need and Ability

Similar to `DWARFService`, the `Ability` and `Need` classes provides access to information about a needs and abilities. `DWARFService` automatically creates `Need` and `Ability` objects when service information is read from the service manager.

Attributes

The `Attributes` class is inherited by `DWARFService`, `Need` and `Ability`. It gives access to additional attributes that may be defined for these entities. `Attributes` is derived from `KeyValueList`, an associative list of string pairs (`std::map<string, string>`).

Session

`Session` provides information about the communication partner, such as hostname and service-id. Session information is always located on the side of the need only.

5.2 Graph Visualization and Layout

GraphView

`GraphView` is a Qt widget which encapsulates the functionality of the `Graph Visualization and Layout` subsystem. It's purpose is to provide a generic Qt component for graph display with automatic layout.

Nodes and edges are added with the `addNode` and `addEdge` methods. `addNode` accepts a unique node identifier string and a list of strings pairs, which define the ports of the node.

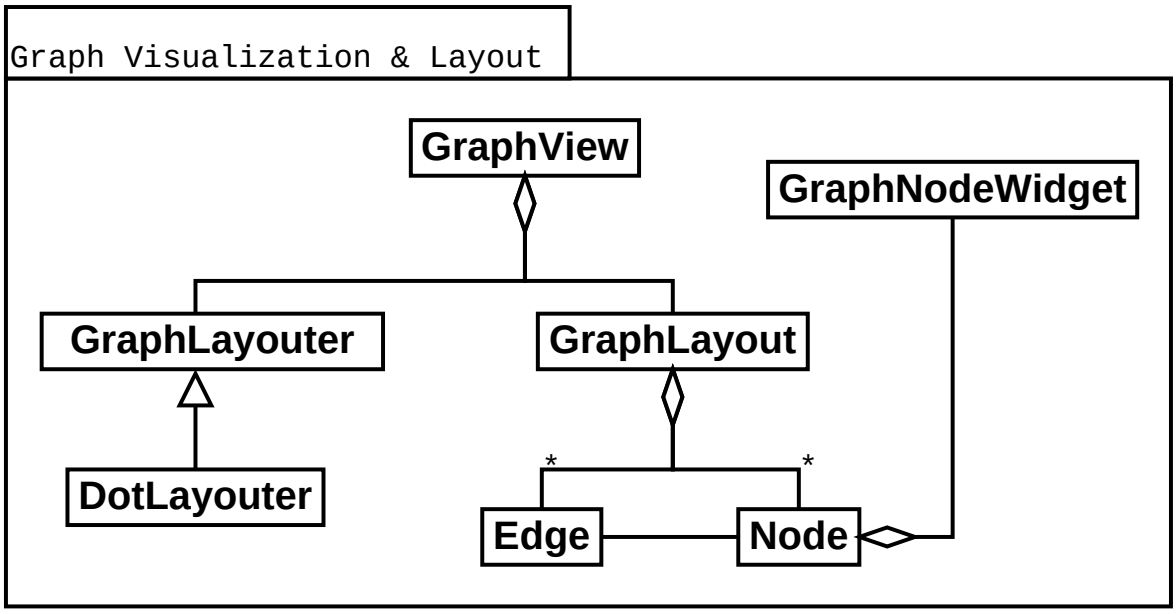


Figure 5.3: Objects of the `Graph Visualization and Layout` subsystem.

The first string in a pair is the port identifier, which should be unique (with respect to the node). The second string gives the text label that is displayed in the user interface. If the node should be visible, at least one port must be supplied, as a node does not contain text labels by itself.

`GraphView` supports user feedback by emitting the `clicked` signal when the user clicks on a node. The signal contains parameters that describe which node and port was selected.

To use the automatic layout facilities, call `Layout`. The `ColorEdges` method can be used to highlight all edges that are connected to a particular node or port. Additional methods include `Clear`, which removes all edges and nodes, and `Print` which writes the graph to a PostScript file if the layouter supports it.

GraphLayout

`GraphLayout` is used internally by `GraphView` to store the graph data structures. It simply consists of two lists of `Edge` and `Node` objects and provides methods to access and modify them.

Edge and Node

The `Edge` and `Node` classes are part of the `GraphLayout` and store the graph information. For efficient access, edges contain pointers to the nodes they connect and nodes contain lists of all edges they are connected to.

GraphNodeWidget

Each `Node` is associated with a `GraphNodeWidget`, which represents the user interface part of a node. The widget is capable of displaying multiple text labels in a UML-like fashion with separators between them. Text labels can be defined by passing a list of `KeyLabel` pairs to the `Update` method. A separator is created when the identifier part of the `KeyLabel` is empty. Texts can be prefixed with a format description. "\b" means bold text and "\c" means that the text should be centered.

When a label is clicked, `GraphNodeWidget` emits a clicked signal with the label identifier as parameter.

GraphLayouter

The `GraphLayouter` class defines an abstract interface for graph layout algorithms. While normal graph layout could have been achieved easily with just one method call, the class defines three methods. The rationale behind this is that, as graph layout might consume a larger amount of time, it should be possible to perform it in a worker thread.

The `Init` method receives a pointer to a `GraphLayout` object and should copy the information from it into it's internal data structures. Then `DoWork` is called which should perform the actual layout work. As it might be called from a worker thread, `DoWork` implementations should be extremely careful when interacting with other components. After the work is finished, the application calls `UpdateGraph`. There the implementation should write the result, i.e. the node and edge positions, back to the `GraphLayout`, which is again passed as a parameter.

DotLayouter

`DotLayouter` implements the `GraphLayouter` interface and calls the dot tool (cf. chapter 3 for graph layout. For communication with dot, `DotLayouter` uses the `IOFilterStream` class which relies on Unix pipes for data transport.

When `Init` is called, the class creates a textual graph description in the dot format. This description is written to stdin of the dot process. `DoWork` simply waits until data is available on dot's stdout port. The resulting graph description with positional parameters in parsed in `UpdateGraph` and written back to the `GraphLayout`. Because the edge positions are described by Bezier splines, the `BuildCubicBezier` helper function is used to convert splines to a number of points with approximately equal distance.

In addition to the methods from `GraphLayouter`, `DotLayouter` has a `Print` method, which causes dot to output the result in PostScript format. The method can be used to export the graph to other applications.

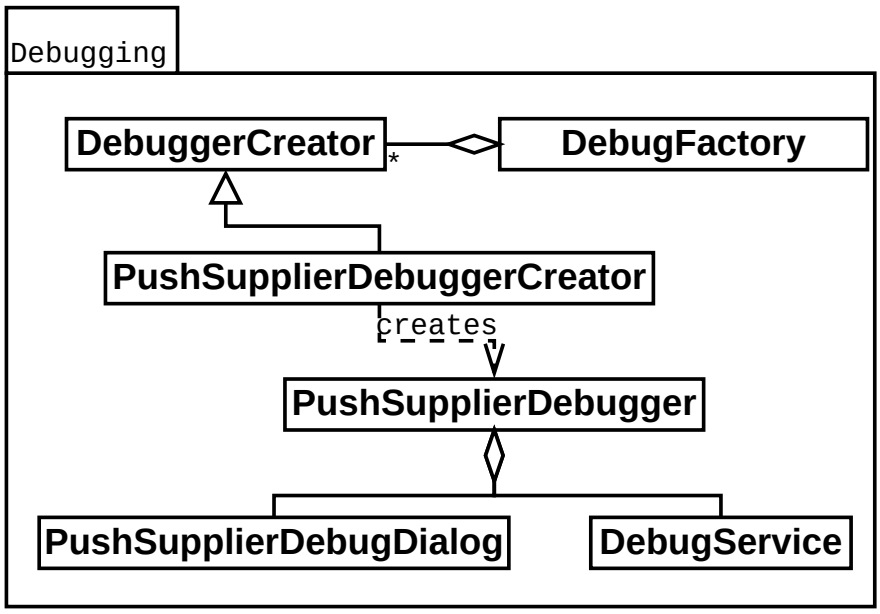


Figure 5.4: Objects of the `Debugging` subsystem.

5.3 Debugging

DebugFactory

The `DebugFactory` is the central part of the `Debugging` subsystem. As the name suggests, the class realizes an abstract factory design pattern that is able to create debugger objects. The interface is divided into two main function groups.

The `FindDebugger` methods are used to query which debuggers are available for the given object. The list returned by all of these methods includes a textual description of the action performed by the debugger (e.g. "View Events") and a unique ID which can be used to create an instance of the debugger. `FindDebugger` methods are available for `DWARFService`, `Need` and `Ability` objects (see section 5.1). The application typically calls `FindDebugger` in order to show the user a list of possible debuggers.

After the application has queried which debuggers are available for the selected service, need or ability, it can call `CreateDebugger` with one of the valid debugger IDs returned by `FindDebugger`. The `DebugFactory` then creates the debugger object and passes it the description of the object it should attach to.

The `DebugFactory` does not decide by itself which debuggers can be used for an object. Instead it delegates that task to one or more `DebuggerCreator` objects.

DebuggerCreator

When implementing a new debugger, the developer must also derive a class from `DebuggerCreator` to register the debugger with the `DebugFactory`. `DebuggerCreator`

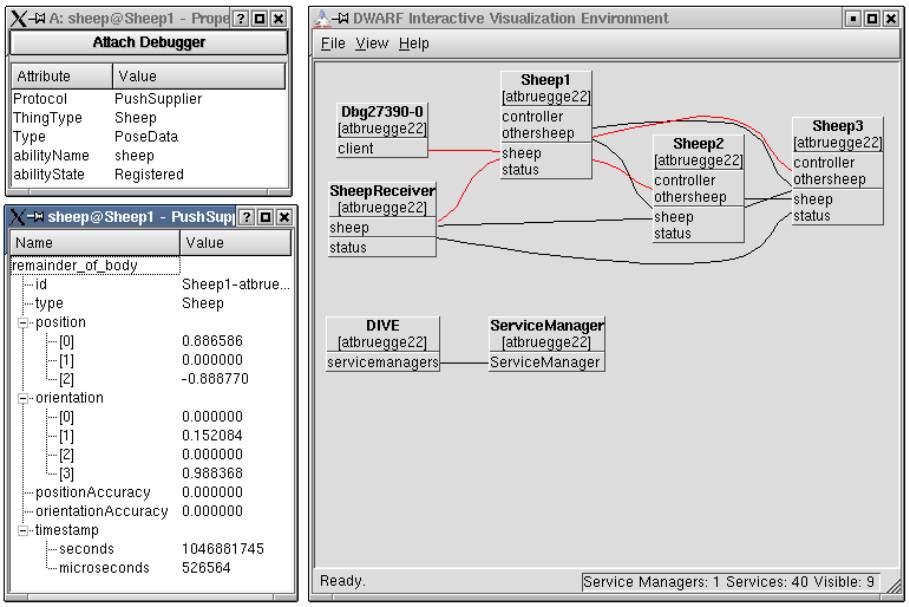


Figure 5.5: DIVE receiveing events from the sheep ability of a Sheep service. The graph shows the `Dbg27390-0` Service created by DIVE in order to attach to the Sheep service.

has `HaveDebugger` and `CreateDebugger` methods for services, needs and abilities in order to determine debugger availability and for debugger creation.

PushSupplierDebugger

`PushSupplierDebugger` is a debugger implementation that receives events from an ability of type PushSupplier. This ability type communicates with other services by sending CORBA structured events using the CORBA notification service [15].

In order to connect to the given ability, the class creates a `DebugService` object, which then adds a new service to the DWARF system and gives it a need of type PushConsumer. By specifying a service ID and an ability name, the predicates of this need instruct the service manager to connect it to only one particular ability. Figure 5.5 shows DIVE debugging a PushSupplier service.

`PushSupplierDebugger` implements the `CosNotifyComm::StructuredPushConsumer` interface. When an event is received, the CORBA notification service calls the `push_structured_event` method. Because this may happen at any time during program exection, `PushSupplierDebugger` packs the event's payload into an `AnyEvent` which is then added to the Qt event queue. The main event loop later delivers the event synchronously to the `event` method where it can safely be processed and displayed.

`PushSupplierDebugger` creates a `PushSupplierDebugDialog` object, which provides the user interface.

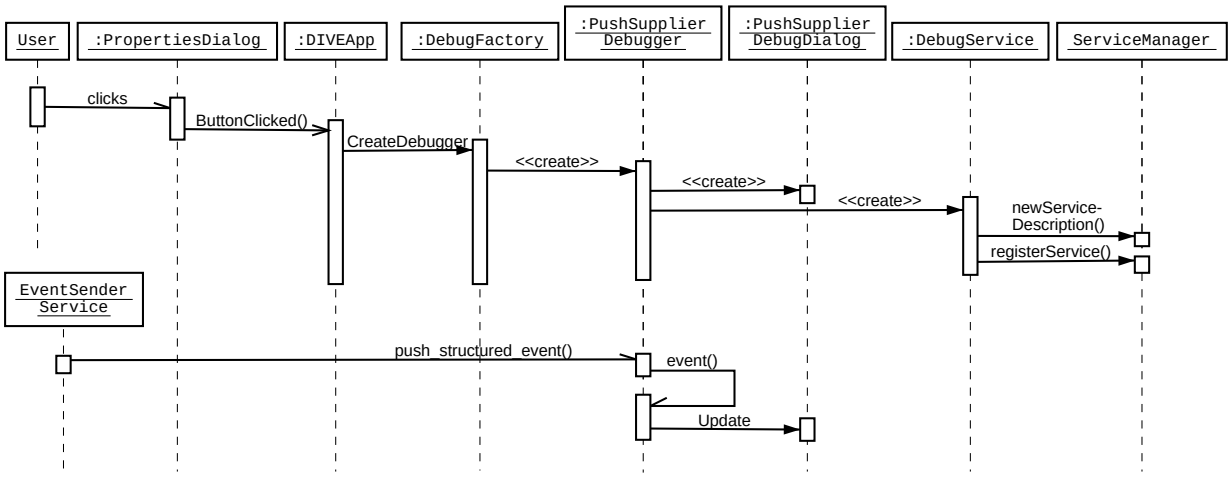


Figure 5.6: Sequence diagram showing how the "MonitorEvents" use case is realized by DIVE. It starts when the user clicks on "View Events" and includes handling of the first event received.

Handling CORBA any values

CORBA structured events consist of a header and a body part [15]. The header has both fixed and optional fields which define domain, type and event name, priority, reliability, etc. The body part has a so-called "Filterable Body Part", consisting of name-value-pairs, and a "Remaining Body Part". While it is possible to use the filterable body fields to store information, DWARF applications typically only use the remaining body part. It may contain almost arbitrary data, so DIVE cannot know at compile time how the event payload is structured.

The "Remaining Body Part" simply is an object of class "any". The "any" type may - as the name implies - contain anything, as long as it's type is known to the CORBA environment. This includes basic data types, such as integers and strings, but also complex structures like arrays, sequences or structs. "any" objects always carry their type information with them. If the receiving application already knows the real type of an "any" object, it can simply cast it and the CORBA libraries perform the necessary conversions. If the type is unknown, things get more complicated.

Fortunately the CORBA specification [14] provides the `DynAny` class for that case. `DynAny` objects can be constructed from an "any" and support operations for type identification and iteration of it's members. The most important methods of the `DynAny` class are described in the following table. For a full documentation of all of it's members and of derived classes refer to the CORBA specification [14].

`TypeCode type()`	Get the object's type.
`Boolean next()`	Move the member pointer forward.
`DynAny current_component()`	Get the current member.
`Boolean get_boolean(), long get_long(), string get_string(), etc.`	Get the content of a basic data type.

Using these basic methods, an application can easily walk through arbitrary data types. The necessary steps are:

1. Use `type()` to check the data type of the object.
2. If the object is a basic data type, use one of the `get_xxx` methods to get it's value. If the object is a composite structure (array, struct, etc.) use the `next()` and `current_component()` methods to recursively query the content.

PushSupplierDebugDialog

`PushSupplierDebugDialog` is the user interface part of the `PushSupplierDebugger`. It has it's own window frame and a list view for the event content. See section 2.6 for a screenshot.

PushSupplierDebuggerCreator

`PushSupplierDebuggerCreator` implements the `DebuggerCreator` interface to register the `PushSupplierDebugger` within the `DebugFactory`.

DebugService

The `DebugService` class creates a new DWARF service with only one need. By defining a predicate, the service manager will connects this need to the given ability. The need name will always be "client". The name of the new service is constructed as follows:

`"Dbg" + <process ID>+ "-" + <counter (starting at 0)>`

5.4 Application

DIVEApp

`DIVEApp` is the main class of DIVE. After an object of this class is instantiated by the `main()` function, it creates a `DWARFSystemModel` and a `DebugFactory` object. `DIVEApp` is derived from Qt's `QMainWindow` class and also provides the main user interface. It creates a new main window with a menu, a status bar and a `GraphView` widget which covers the rest of the window.

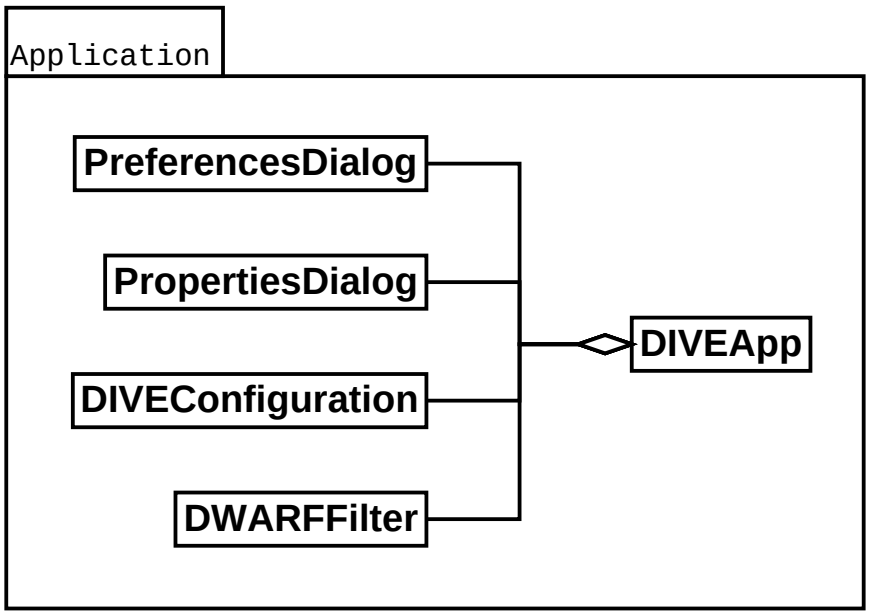


Figure 5.7: Objects of the `Application` subsystem.

`DIVEApp` implements a mediator pattern between the other subsystems. When the user selects "Update" in the menu, the information from the `DWARFSystemModel` is translated into a graph structure for `GraphView`. Similarly, when a service is clicked in the graph, `DIVEApp` reads it's extended attributes from the system model, asks the `DebugFactory` about available debuggers for that service and then creates a new `PropertiesDialog` to show that information to the user. The mediator behaviour of `DIVEApp` is detailed in figures 5.6, 5.8 and 5.10.

DWARFFilter

`DWARFFilter` is used by `DIVEApp` in order to decide if a given service should appear in the graph. The only supported method is `bool operator()` which returns true if the service should not be displayed. `DWARFFilter` does read it's parameters directly from `DIVEConfiguration`.

The current implementation can only filter unregistered and template services, but future versions should be extended to allow filtering of services by their attributes.

PropertiesDialog

The `PropertiesDialog` class implements a Qt window which contains a list view and an arbitrary number of buttons. It is created by `DIVEApp` when the user clicks on a service, need or ability in the graph view. The `ButtonClicked` signal is emitted when one of the buttons is clicked. In order to change the contents of the window, use the `Update` method.

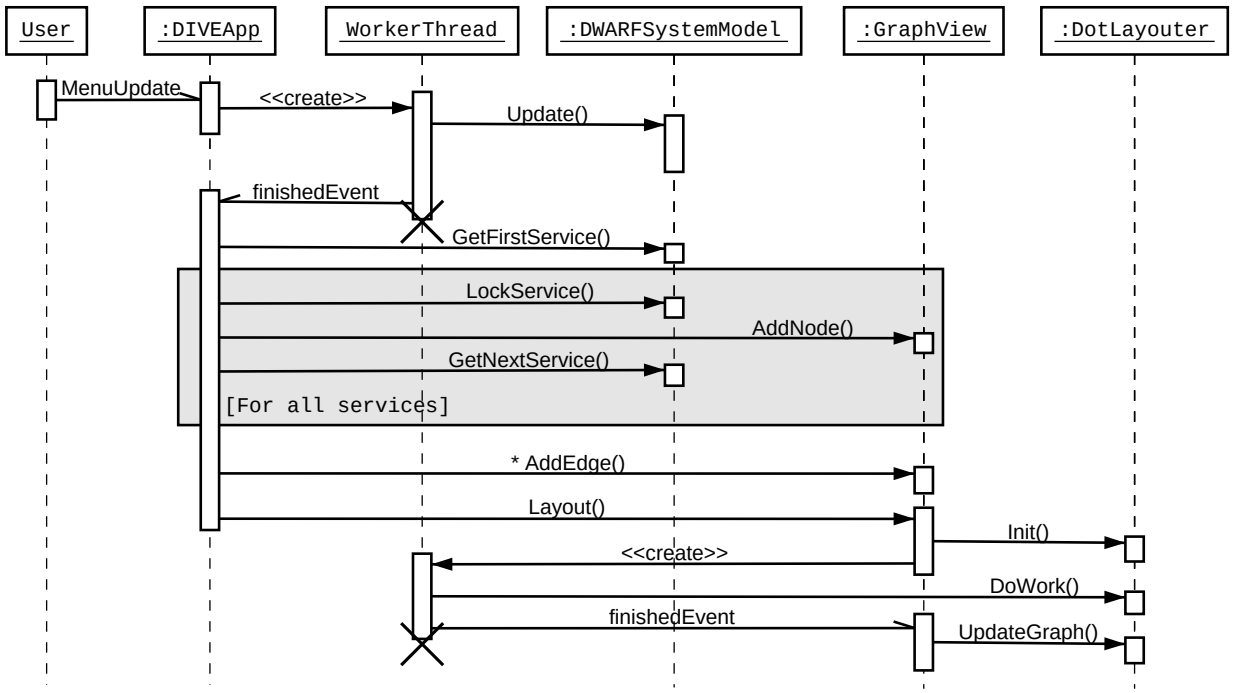


Figure 5.8: Realization of the "Update View" use case. Updating the `DWARFSystemModel` was detailed in figure 5.2.

PreferencesDialog

The `PreferencesDialog` is a dialog with controls for all user preference settings. It is displayed by `DIVEApp` when the user selects "Preferences..." from the menu.

DIVEConfiguration

`DIVEConfiguration` is derived from `CConfiguration` and contains members for all user preference settings that need to be stored persistently. The values are loaded by `main()` at program startup and saved at shutdown.

5.5 Auxiliary Classes

The auxiliary classes do not belong to a particular subsystem. The are used by various subsystems for specific tasks. All of the classes are very general, so they can be reused in other projects without change.

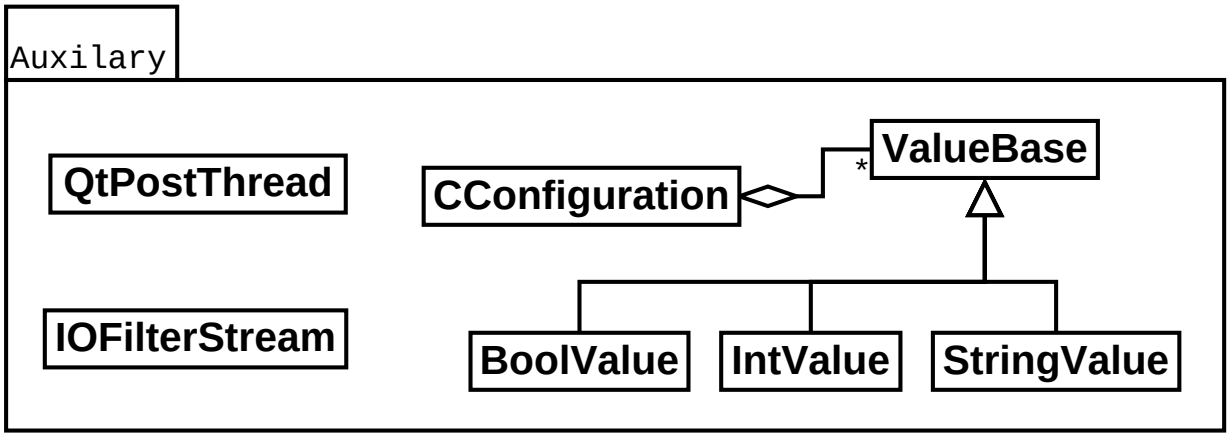


Figure 5.9: Overview of the auxiliary classes.

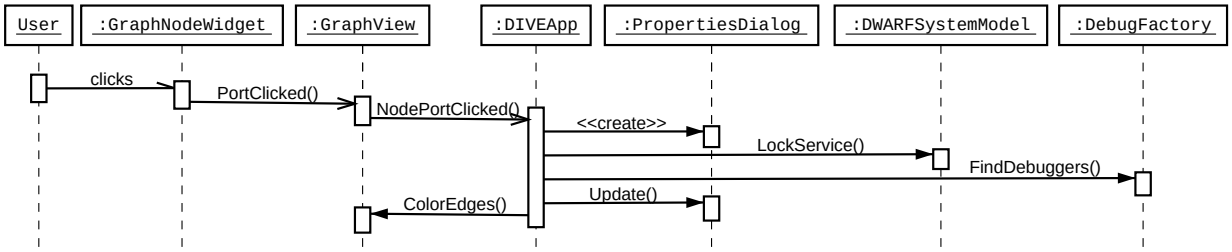


Figure 5.10: Interactions of DIVE's components when the "Select Service" use case is perfomed.

QtPostThread

One major obstacle when implementing worker threads is that the programmer usually has to provide a static function that again calls an object's method or - if a thread abstraction library is available - has to derive the thread from a particular "runnable" class, which limits portability. `QtPostThread` simplifies this task. It is based on C++ templates and can call any method with a void `xxx()` signature of any class. When the thread has finished, an optional asynchronous Qt event can be sent to a given object.

`QtPostThread` is used by `DIVEApp` when updating the system model and by `GraphView` for asynchronous graph layout. Figure 5.8 gives more details.

IOFilterStream

`IOFilterStream` runs another program and makes it's stdin and stdout ports available to the application. The normal C++ streaming operators ("<<" and ">>") can be used to write and read data to or from the program. `IOFilterStream` does this by implementing it's own streambuf class. A description of how to do this is found in [17].

CConfiguration

`CConfiguration` is a base class for user preference data that has to be stored persistently. It's main advantage is that all member variables are automatically saved and loaded, when the `Save` and `Load` methods are invoked.

`CConfiguration` already provides three subclasses for popular data types: `BoolValue`, `IntValue` and `StringValue`. Other types can be derived from `ValueBase`. The current implementations use Qt's `QSettings` class in order to save and load the values to a configuration file. For all types a default value can be specified which is used when no previously saved value is available.

For an example of how to use `CConfiguration`, have a look at the implementation of the `DIVEConfiguration` class.

Chapter 6

Future Extensions

At it's current state, DIVE already is a valuable tool for development, debugging and presentation of DWARF systems. But there still are a lot of possible extensions which - hopefully - will be implemented in the future.

The purpose of this chapter is to propose future extensions of DIVE. To facilitate the development, I also will give some hints about how these features could be integrated into the current design.

6.1 Other Debuggers

In order to add new debuggers to DIVE, a new class has to be derived from `DebuggerCreator`, which performs availability testing and creation of the debugger. An instance of this class must be created in the constructor of `DebugFactory`. In case the new debugger has it's own windows, a Qt parent object is provided when `CreateDebugger` is called.

To simplify connecting to a need or ability, the `DebugService` class can be used, which creates a new service with the right predicates. Only the protocol-dependent interfaces have to be implemented by the debugger.

Here are some ideas of future debuggers:

CORBA Method Invocations

Together with CORBA structured events, CORBA remote method invocation is one of the two most frequently used communication forms in current DWARF applications. Therefore a debugger for this protocol seems to be the next thing to do.

As a method invocation only goes to one receiver, monitoring the call is not directly possible. A solution is to insert the debugger service as a proxy between the two services. This probably will require support by the service manager, but then all calls can be intercepted using the dynamic skeleton interface for receiving and the dynamic invocation interface for sending of messages. This works even if the signature of the methods are

unknown at compile time. Dynamic skeleton and invocation interfaces are described in [14]. A completely different approach is using an orb with logging capabilities.

A different way of using CORBA method invocations at runtime is by sending them to a service which exposes one or more interfaces. For this purpose, again the dynamic invocation interface can be used. As the interface structure in general is not known at compile time, a CORBA interface repository is needed. Unfortunately the OmniORB does not provide one. A solution is to use the interface repository of one of the Java ORBs. Alternatively, the DISTARB tool by Marcus Tönnis can be started as an external debugger (see below).

External Debugger

Not all debuggers necessarily need to run inside the DIVE application. Especially for the more complex ones (e.g. image stream viewers), running them in their own process seems reasonable. Such a facility also allows easy integration of existing tools like DISTARB or ManualTracker.

To implement this, some kind of configuration file is necessary. This can be an XML file with a description of all external debuggers. The contents of this file should include the path to the executable, the program arguments (with wildcards for service id and need or ability name), the name of the button in the properties dialog and the preconditions (protocol, attributes) that have to hold before the button is displayed. For each external debugger found in the file, an object derived from `DebuggerCreator` must be added to the `DebugFactory`.

Logging in Event Debugger

This describes not a new debugger, but rather an extension of the existing `PushSupplierDebugger`. The current implementation only displays the most recently received event, but for some event types also the history is important. Therefore the `PushSupplierDebugDialog` should be extended with a QTabWidget that provides a second page which has a text view where all events are logged. In order to convert an event into a line of text it is not necessary to use the `DynAny` facility again - the event structure easily can be read from the existing tree view.

6.2 Other Views

An obvious extension of DIVE is replacing the two-dimensional graph display with a different interface. The Qt widget that occupies the largest area of the DIVE main window can be replaced easily. Unfortunately those parts of `DIVEApp`'s code which translate data from `DWARFSystemModel` to `GraphView` need to be replaced as well. This opportunity should also be used to move this code into it's own class.

Some ideas for future views are:

List View

This is a very simple view that presents a list of all services. Connections between the services are not visible.

3D View

As DWARF is intended for augmented reality applications, a three-dimensional view of the graph seems natural. Fortunately, Qt provides a OpenGL widget which can be used for that purpose. In order to store the graph and to perform the layout, the GraphLayout and GraphLayouter classes can be reused, but have to be slightly modified as they currently rely on Qt Widgets to represent the nodes.

An alternative approach is to use the VRML output capabilities of Graphviz in a way similar to the existing export function. The resulting file is then sent to an external VRML viewer. The quality of Graphviz's VRML output however has not yet been tested.

Edge Labels

This is an extension of the current 2D graph view. Graphviz is capable of placing a text string next to each edge in the graph arrangement. This can be used in order to add e.g. the protocol name to each connection. The implementation is relatively easy but there is doubt if it is useful at all, because current DWARF system graphs already contain a lot of connections and adding edge labels probably will contribute more confusion.

6.3 DWARF System Design with DIVE

The obvious approach to a graphical design of systems which consist of existing components is the classic visual programming one. The designer can instantiate a component by choosing one from a palette and placing it on the screen. Then the objects' communication ports are connected by drawing lines with the mouse. A good example for this approach is the "Visual Network Editor" of AVS [7], a system used for processing and visualization of scientific and business data.

Such a visual programming method however contradicts the self-assembling nature of DWARF systems. In particular, their ability to choose the best communication partners with respect to the context is reduced. Therefore, for DWARF a different approach has to be found - but a complete discussion of this topic would exceed the scope of this document.

Manipulate Service Descriptions

Rather than specifying connections between services directly, an approach that seems to fit better for DWARF systems is manipulating the attributes of existing services such that the system assembled by the middleware has the desired properties.

The user interface of such a manipulation mechanism can be easily integrated into DIVE as a debugger (see section 6.1). This makes the manipulator available as a button in

the properties dialog. If the manipulator needs access to the service manager's interfaces, the `DWARFSystemModel` should be extended to provide this.

A service that is being modified is in an unfinished intermediate state and usually should not be connected by the service manager until it is explicitly released. Such a service, however, should be displayed in the graph view. One interesting question is where the information about this service resides. In theory, the service description can be placed in the service manager with an additional attribute that keeps the middleware from connecting it. When the view is updated, the service description is read by the `DWARF System Model` subsystem and displayed in the graph view. Alternatively, unfinished service descriptions can be stored inside DIVE in the `DWARF System Model`. When editing is finished they need to be transferred into the service manager.

The current middleware already supports template service descriptions and start-on-demand services. Using these mechanisms to create new services should be considered.

Remote Service Configuration

Many existing DWARF services have some kind of internal parameters (e.g. speed of a sheep) that can be changed by the user. This is done either by specifying the parameters at the command line or - if modification at runtime is necessary - by developing a custom user interface, which requires a lot of extra work. Both approaches only allow parameter modification from the console where the service was originally started.

Service configuration can be simplified by supplying the service with a simple interface which allows others to find out at runtime what parameters are supported and what their current value is. Of course, changing these parameters is also possible.

Such an interface can be used to create another debugger for DIVE which reads the current parameters, opens a window with a control for each of them and allows the user to change the values. This permits easy reconfiguration of services at runtime.

Saving the DWARF Application

Both features described above - manipulation of service descriptions and remote service configuration - together allow a developer to build new applications using existing DWARF services. What still is missing is a way to store the changes persistently and reload the whole application later. Details of how this can be done is the subject of further research by the DWARF team.

Manual Layout

A feature that can be important for many kinds of system design is a layout engine which lets the user place services manually. Ideally such a layouter would be capable of keeping manually arranged services in place while the position of the others is determined automatically. The neato layouter seems to support pinning nodes to fixed positions, but unfortunately ports do not work yet (cf. chapter 3).

The first step towards an implementation necessarily is the addition of incremental updates (see section 6.4), as the current situation is not acceptable, where the complete layout is discarded at each update.

The other modifications should only affect the `Graph Visualization and Layout` subsystem. `GraphNodeWidget` must be extended with a drag-and-drop ability and a layouter has to be implemented which does not change the position of the manually placed nodes.

6.4 Dynamic and Incremental Updates

Currently, when an update is performed, the whole content of the `DWARF System Model` subsystem and the `GraphView` is discarded and rebuilt. This is somewhat unsatisfying because there are a number of disadvantages:

- DIVE does not know if the system has changed at all and repaints the graph often. This results in increased system resource usage and visible flickering.
- If an incremental layouter (cf. chapter 3) was available, it could not be used because the graph is drawn from scratch, without knowing the previous positions.
- The current experience shows that sometimes communication between DIVE and distant service managers can be very slow. Therefore the updating process can only be repeated in large intervals, which negatively affects the response time to system changes.

The solution is to update only those parts of the graph that have changed. Ideally this is done with the help from the service managers. Whenever the state of a service has changed, they could notify DIVE, which only then would update the display. This could also dramatically improve response time.

But even without help from the service manager, it is possible to limit screen flickering and invocation of the layouter. After doing an update, the `DWARFSystemModel` compares the new system state with the previous one. If they are equal, no redrawing is necessary. Also the update could be performed in multiple threads, so slow service manager connections will no longer affect the overall response time.

For the implementation of incremental updates, `GraphView` has to be extended to allow modifications of the graph without completely discarding the previous state. The necessary functions must allow modification of nodes (change ports) and removal of nodes and edges. Also, `DWARFSystemModel` needs to get an interface to communicate service changes. This could be done by some kind of callback mechanism, based on Qt signals, or by polling in short intervals. To describe the system changes, a list with the ids of the changed services should provide a reasonable level of granularity.

Chapter 7

Conclusion

The previous chapters have given an overview of the functionality and the internal structure of the DWARF Interactive Visualization Environment.

The focus of Chapter 2 were the requirements for a visualization tool for DWARF systems. The functionality of DIVE was described from the a user's perspective. This description included use cases and screen shots. Chapter 3 gave an overview of graph layout in general, described various existing tools and showed why finally Graphviz was selected for the first version of DIVE. Chapter 4 described the system design. We saw how DIVE is devided into four subsystem and how the main design goals - extensibility and reusability - are achieved. The actual implementation was described in chapter 5. It gave an overview over each class and showed how DIVE handles some interesting technical tasks. Chapter 6 focused on the future of DIVE. Despite of it's current usefulness, there still are many extensions which could make DIVE even more valuable. A number of additional features were proposed. The chapter also explained how these features can be integrated into the current design.

The current version of DIVE works reasonably stable and has proven to be a valuable tool for the demonstration and development of DWARF systems. It was used in order to publicly demonstrate the system design of the SHEEP system [16] and to explain the functions of the DWARF middleware at the ISMAR2002 conference and at various other occasions. DIVE is also actively used in the development process of current DWARF applications, such as ARCHIE [19].

More research has to be conducted on what tools and processes are necessary in order to allow an easy development of new DWARF applications out of existing services. One interesting question is how such an application can be stored permanently in order to launch it at a later time. As we have seen in chapter 6, DIVE can be extended to partly achieve these goals, making it an even more useful tool in the development process of DWARF applications.

Bibliography

[1] DWARF *Project Homepage*. Technische Universität München, `http://www.augmentedreality.de`.

[2] *aiSee Product Page*. AbsInt, `http://www.absint.com/aisee/`.

[3] *AGD Homepage*. TU Wien, Uni Köln, MPI, `http://www.ads.tuwien.ac.at/AGD/index.html`.

[4] *LEDA Product Page*. Algorithmic Solutions, `http://www.algorithmic-solutions.com/enleda.htm`.

[5] *Graphviz Homepage*. AT&T, `http://www.research.att.com/sw/tools/graphviz/`.

[6] *VGJ Homepage*. Auburn University, `http://www.eng.auburn.edu/department/cse/research/graph_drawing/graph_drawing.html`.

[7] AVS *Web Site*. `http://www.avs.com/`.

[8] 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.

[9] *daVinci Presenter Homepage*. b-novative, `http://www.davinci-presenter.de/`.

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

[11] *Graph Editing Framework Homepage*. `http://gef.tigris.org`.

[12] *JViews Product Page*. ILOG, `http://www.ilog.com/products/jviews/`.

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

[14] Object Management 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.

[15] Object Management Group, *Corba Notification Specification.* `http://www.omg.org/technology/documents/vault.htm#svc_and_fac`, June 2000.

[16] C. Sandor, A. MacWilliams, M. Wagner, M. Bauer, and G. Klinker, *SHEEP: The Shared Environment Entertainment Pasture*, in Demonstration at the IEEE and ACM International Symposium on Mixed and Augmented Reality – ISMAR 2002, Darmstadt, Germany, 2002.

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

[18] *Graph Layout Toolkit Product Page.* Tom Sawyer Software, `http://www.tomsawyer.com/glt/index.html`.

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

[20] *Graphlet Homepage.* Universität Passau, `http://www.infosun.fmi.uni-passau.de/Graphlet/`.

[21] *GDToolkit Homepage.* Università di Roma Tre, `http://www.dia.uniroma3.it/~gdt/editablePages/main_index.htm`.

[22] *VCG Homepage.* Universität des Saarlandes, Saarbrücken, `http://rw4.cs.uni-sb.de/users/sander/html/gsvcg1.html`.