Umeå University
Department of Computing Science

Master's Thesis in Computing Science

Architecture of Plug-in Interfaces

Samuel Tärestam

Abstract

Plug-in architecture is today one of the most popular techniques used to increase the user customization of programs. The architecture allows developers to write plug-ins of their own to extend a program with new functionality. This thesis presents a few techniques such as inheritance, metadata, middleware etc. that can be used to implement the interface that enables the plugging and unplugging of plug-ins. The results of this thesis are suggestions to which plug-in interface to use depending on the requirements on a program. Besides this investigation of interfaces, the development of Estrutura, a program using plug-in architecture to solve a real problem is presented. This thesis gives a description of Estrutura that together with the description of interfaces can be used if a new plug-in based system is developed.

Samanfattning

Plug-in-tekniken är en av de populärare teknikerna för att öka en användares möjlighet att anpassa ett program. Plug-ins gör det möjligt för utvecklare att skriva egna plug-ins för att skapa ny funktionalitet i ett program. Denna rapport presenterar ett antal tekniker t.ex. arv, metadata, middleware m.fl. som kan användas för att skapa gränssnittet som möjliggör att plug-ins pluggas i och ur. Resultatet av rapporten är förslag på vilka plug-in-gränssnitt som är lämpliga beroende av kraven på ett program. Förutom undersökningen av gränssnitt presenteras utvecklingen av Estrutura vilket är ett program som använder plug-in-tekniken för att lösa ett verkligt problem. Detta examensarbete ger en beskrivning av Estrutura som tillsammans med beskrivningen av gränssnitt kan användas för att underlätta designval vid utvecklingen av nya plug-in-baserade system.

Preface

This thesis is mainly intended for software developers and system designers in need of techniques to create dynamic and extensible software systems. Some answers will be presented on how a plug-in architecture can be implemented and which choices that can be made during the design of such a system.

The problem this thesis investigates relates to the area of software engineering within computing science. The original proposal of the work was formulated together with TietoEnator in Umeå at their business unit for "Telecom and Media", division "Radiosys".

Examiner and instructors

The following people beside myself have been a part of carrying out the work or the examination.
Examiner, Department of Computing Science: Per Lindström
Internal instructor, Department of Computing Science: Johan Karlsson
External instructor, TietoEnator - Telecom and Media: Gunnar Strand

Contents

List of Figures

Chapter 1

Introduction

There are today several topics within the research area of software engineering related to techniques that allow programs to be extensible and dynamic in different ways. One of the techniques uses whole programs or parts of programs as pluggable software, enabling the software component to be plugged into another program to become a part of it. This technique is used to allow developers building their own extensions to existing programs but also to allow software companies to release updates to their products as plug-ins rather than to update the whole program. The concept of programs that are pluggable will be described in depth in this thesis and presented as "plug-ins". Programs allowing plug-ins to extend its functionality will be described as host programs.

In this thesis the concepts plug-in and host program will be investigated. An example of a host program that supports plug-ins will be described in detail as well as different interfaces used to enable plug-ins. This will give an example of how a certain problem can be solved using plug-ins and also present some suggestions of improvements to future implementations. A theoretical study of the interface, between a plug-in and a host program, which enables plugging/unplugging the plug-ins has also been carried out.

Current topics within this research area and other similar areas, trying to solve the same type of problems, will be reviewed in the end of this thesis together with some reflections.

Chapter 3 will investigate the plug-in concept, how it is related to software engineering and how other similar concepts are related to plug-ins. An indepth investigation on plug-in interfaces will also be presented as well as a discussion on when a certain interface is appropriate.

In chapter 4 a description of the problem will be presented along with the situation before this thesis.

The example host program that will be presented in this thesis will together with a few plug-ins be used to support the software engineering at TietoEnator in Umeå. The process of developing a host program including collecting and describing requirements, implementation and evaluation will be covered in chapter 5. The host program and a conclusion from the results of the investigation of interfaces will be presented in chapter 6.

Chapter 2

Background

An oftenly discussed topic within software engineering is reuse of code without resorting to the copy and paste technique. There are several different research areas trying to find ways of making programming efficient. Some techniques used are patterns, component based modeling, aspect oriented design and dynamic software architectures [20, 19].

The investigation described in this thesis originates from TietoEnator, Telecom and Media in Umeå who needed a tool that was easy to extend and would enable them to easily develop plug-ins of their own.

The following description is the information out of TietoEnator's point of view:

> "Cello is the platform on which the radio base station software developed at TietoEnator Telecom & Media in Umeå is based. There exists neither environments for executing Cello programs, nor for analyzing trace logs generated by Cello programs. Almost all designers, basic testers etc. manually start separate windows to run the program and to see the traces.
>
> Control commands to the Cello program are manually inserted on the command prompt using copy paste from a text editor into the window where the program is executing. There is no integrated support for sending commands to the program during execution.
>
> Generated traces are viewed in a separate window with no navigation or analysis support. There do exist some stand-alone programs which can perform color coding of the log output, but no tool so far have any heuristics at all ." [39]

2.1 Thesis Proposal

With this background it was proposed that a generic host program should be developed to support plug-ins that are able to execute Cello programs and analyze the trace logs which are generated.

The theoretical study should investigate how a system supporting plug-ins could be designed, how this system can be tested and whether a similar system

already exists that can be used as a host for the considered plug-ins. The result of the investigation should suggest an existing system or conclude with an implementation of a system. If possible given the time restraint, test environment and plug-ins that support analysis of traces and execution of Cello programs should also be provided to the system.

Chapter 3

Plug-ins

Some of the more theoretical parts of this investigation will be described in this chapter where both the context will be described as well as an in depth look at different techniques on how to enable a plug-in to connect to a host program.

The questions on how this research contributes to the computing science will be described in 3.1. And what it is that makes a plug-in a plug-in and not a module, component or even a separate program is discussed in 3.2. In 3.4 an in depth description on different techniques to implement the interface between a plug-in and a host program will be described. Some of the existing solutions for systems using plug-ins will be described in 3.5, and in 3.6 a point to related work will be made.

3.1 Software Engineering

The research within software engineering is to this date a research area that expands and branches out in topics that might become separated from the software engineering genre someday. Two of the branches in software engineering are reusable components and extensibility of code or function [42]. These two branches are both subject to research on plug-ins. A plug-in could be seen as a component and it could also be seen as an extension to a program that adds some additional functionality.

3.1.1 Reusable components

The *component* concept in the area of software engineering refers to a separate binary object. A deeper clarification on what a component is, will be presented in section 3.2. Some believe that the purpose of *reusable software components* is to create a component market where the components are independently created and making the software extensible, this would possibly save time and money for the developer [46].

When considering *reusability* within software engineering there are several different types of things to reuse. The two types from the developer's point of view is reuse of components without modification (known as black-box reuse) and modification of components to satisfy the reuser's particular needs (called

clear-box reuse).

A problem within object oriented programming is that even though there exists working and tested components, many of the new applications are written from scratch. The market for components is not as large as it could be. This is because the components are often developed by many different organizations, they have no standard interface, and are often integrated into programs making them tightly coupled [17]. Even though there exists some problems in distribution of components and making them fit into new programs, there are several things to gain from reusing software components allowing users to develop their own components to extend the functionality of a program [18].

3.1.2 Extensibility of code or functions

The other topic within software engineering that should be considered when looking at plug-ins and components is *extensibility*. Extensibility is often related to components and 3rd party developers. There are several types of extensibility: programs that can be extended with a patch, plug-ins that can be loaded during runtime providing new functions and enhancements [8], and programs that dynamically change themselves during runtime [40].

Some developers use the extension concept when they are referring to, what others would refer to, as a plug-in [27, 28]. This situation, where the concepts are mixed up, can be confusing but it also tells us that a plug-in is something that extends the functionality of a program, more on this in section 3.2.

3.1.3 Contribution

Since software engineering consists of both components and extensibility, the contribution plug-ins makes to software engineering, is a mixture of these two with some new aspects.

The plug-in concept is not a commonly used concept within the research of software engineering. There are several different areas where research is focused on a number of properties included in the plug-in technology, such as components, extensibility and dynamic architectures, but not one separate area is focused on plug-in architecture. Since there is no clear description on what a plug-in is, and how it is related to existing research areas within software engineering, section 3.2 will investigate this relationship and give a definition of what a plug-in is.

Most of the research and existing programs found on plug-ins has implemented "a plug-in architecture" without a motivation or investigation on why a certain interface is chosen between the plug-in and its host program [21, 27, 28, 7, 8]. There is no known investigation on the interface between a plug-in and its host program, an investigation on this subject will be presented in section 3.4. Section 3.4 will also present different types of interfaces that can be used

connecting a plug-in to a host program.

3.2 Plug-in, Module, Component or Program?

So far a the *component*, *module* and *plug-in* concepts have been used within software engineering in a way that can make anyone confused on what it is that separates them [27, 7, 8]. This part of the thesis will try to separate the different concepts when it is possible. The following sections will not try to give a full definition of each concept but rather try to find out what it is that separates them and how they are related.

The following sections will each introduce a new concept and discuss how it is related to plug-ins. Other relationships, eg. between an extension and an API will not be covered here.

3.2.1 Plug-in

A *plug-in* is a software unit that gives the host program it is plugged into, some enhancements or new characteristics. To describe a plug-in and what it is, some of the concepts presented in the following sections can be used.

The properties that together separates a plug-in from the other concepts are:

- it is a program; it consists of instructions on how it should be executed,
- it is a component; every plug-in is a separate piece of software with a defined interface,
- a plug-in is able to be plugged into and out from a host program during runtime,
- a plug-in can offer services used by the host program or other plug-ins, and
- a plug-in can depend on and use services and resources provided by the host program or other plug-ins.

The plug-in technique provides modularity in the sense that every plug-in is a separate component with its own functionality or characteristics [33].

If a plug-in provides services and is depending on services, then there can possibly be some problems with circular dependencies. Figure 3.1 shows an example of circularity which could be a problem. If the services are managed by the host program, then it is up to the host program to decide when services can be accessed and when a service provider can be changed.

One problem that could occur if there exists circular dependencies, is the possibility for infinite service loops. The plug-ins are trying to use each others

services without any condition for termination, this implies a risk for "dead-lock". The problem with circularity is not an issue to be solved here, only presented to make the potential problem visible.

The definition above of a plug-in is the one used when the interface between a plug-in and the host program is investigated.

3.2.2 Program

A computer *program* is something that can be executed by a computer. The requirements on a set of data to call it a program differs between people and computers. The definition of a program presented below is good enough to be used when investigating the relationship between a program and a plug-in, which is the purpose of this section.

> Program: "An organized list of instructions that, when executed, causes the computer to behave in a predetermined manner." [44]

The specific thing which makes a program a program and not a plug-in, is extra properties included in the requirement of plug-ins. A plug-in is dependent on a host program and must be able to be plugged/unplugged during runtime of the host program. The similarity between a plug-in and a program is that they both fulfill the definition of a program. A computer program is not necessarily a plug-in but according to the definition above a plug-in is a computer program. The definition given above makes many different types of files candidates for being computer programs. For instance, a script[1] could be called a program depending on how the program concept is defined.

[1] "Another concept for macro or batch file, eg. a script is a list of commands that can be executed without user interaction" [45]

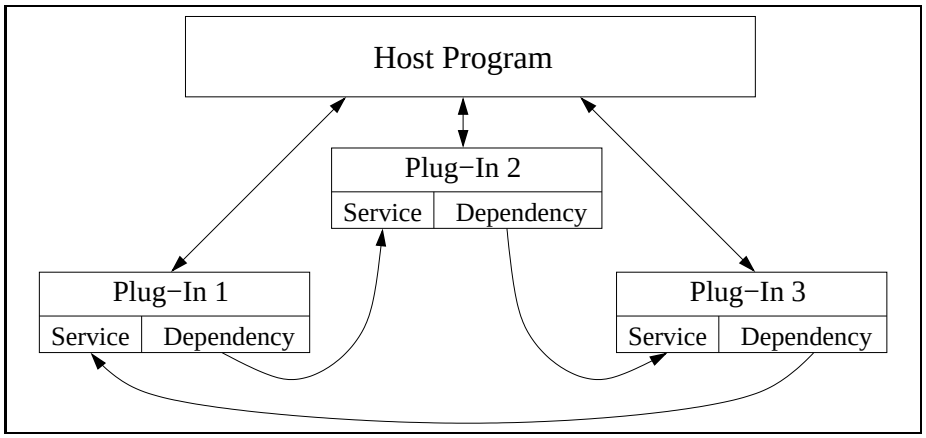

Figure 3.1. Circular dependencies of services.

As a conclusion to programs and their relationship to plug-ins, a program can be a plug-in if there exists an interface in a host program that can execute the program during the host program's runtime.

3.2.3 Component

The *component* concept is probably something that most people know what it is, but a component as it is used within software engineering is somewhat ambiguous. The definition differs and can sometimes be used as a synonym for plug-ins and sometimes as a building block of a program with other requirements than a plug-in. When a component is used to refer to a part of a program, the component is considered to be a composite part with some functionality where the input and output, or characteristics are well-defined [33, 15].

The functionality of a component can be anything from a high-level function to a low level technical task like "read data from disc". An important property of a component is, it can be atomic or composite. An atomic component is totally standalone and independent. A composite component consists of subcomponents that provides lower level functionality, this composition of components can lead to an hierarchical architecture of components [8, 33].

The common theme when components are discussed is to view them as separate from each other and that all interaction with a component is made through the interface defined by each component [35]. There is nothing wrong to say that every plug-in is a component, since a plug-in is a separate software component with an interface of some sort. The opposite relationship, that all components are plug-ins, would not be accurate. This makes a plug-in a bit more than just a separate component with an interface. The reason why all components cannot be seen as plug-ins is that a plug-in is written to match the interface provided by a host program. A plug-in must also be able to be plugged and unplugged during runtime, specified and controlled by that interface.

Components are often mentioned together with the reuse concept because the reusable piece of software is often a component. Sometimes reuse of components could even increase a system's performance and reliability because of prior testing and optimization [33, 17].

Since the requirements on a component are fewer than on a plug-in, the component concept should be seen as a more general concept.

3.2.4 Module

A *module* and the *modularity* concept is tightly connected to the previous section on components. Even though the relationship between a module and a component are not to be discussed here, it is difficult to describe a module without comparing it to a component. The module concept is often used when a component is considered and vice versa. When a system is considered to be

modular it is said that "each activity of the system is performed by exactly one component, and only when the inputs and outputs of each component are well-defined" [33].

To describe a module as something different than a component would be incorrect since the module and component concepts are used as synonyms depending on the author [33, 46, 19, 16].

3.2.5 Extension

As the name tells us, an *extension* is used to give new functionality to a system or to give it some new characteristics by adding an extra software component.

Some programs use the extension concept in the same way as a plug-in. Below is an excerpt from the description of extensions to the application Mozilla Firefox:

> "What is an extension? Firefox Extensions is the name for various enhancements to the browsing experience in Firefox. They can be thought of as small programs (or add-ons) that add new functionality to Firefox.
>
> The great thing about extensions is that they allow Firefox to stay small and unbloated. Anyone who require more features can download the appropriate extension." [11]

As we can see the extension concept is used in the same manner as the plug-in concept. The properties described in the quote might as well have been about a plug-in. One difference compared to the previous definition of a plug-in is that an extension does not have the requirement to be able to load during runtime of the host program. Extensions are very similar to plug-ins and require no further investigation at this point.

3.2.6 Discussion

As we have seen in the previous sections, the concepts are related and it is not difficult to understand that they are mixed up on occasion.

The concepts considered above are used both in software engineering [33] and actual applications [11]. Some of the concepts not discussed here such as package, library, container, function and add-in, are concepts that, depending on their context, can be related to plug-ins. The reason for not covering them here is that the purpose of this chapter is to clarify what a plug-in is, and for this, not all concepts need to be taken in consideration, only those most closely related.

The plug-in concept defined above is very general and could make a program running in an operating system a plug-in to the operating system (where the OS

would be synonymous to a host program). To limit the definition of a plug-in to a precise definition would not make much sense since the plug-in concept is used in many different ways and contexts [28, 37, 5, 26]. The properties of a plug-in used in this thesis will, even if they are general, probably not cover everything called plug-ins today. Using a more general definition of a plug-in would work but it would also make it harder to separate a plug-in from the other concepts presented above. The bottom line is that the definition given above will be good enough for the examples of programs presented in the section on interfaces, and it will also present the main ideas of what a plug-in is all about.

3.3 Host Programs

A *host program* is an ordinary computer program written in a language that supports loading of classes, packages, modules or even standalone programs during runtime. It supports plugging/unplugging during runtime.

The host program might provide the plug-ins with resources needed or requested, and these resources may not only be from the host program but also from other plug-ins managed by the host program.

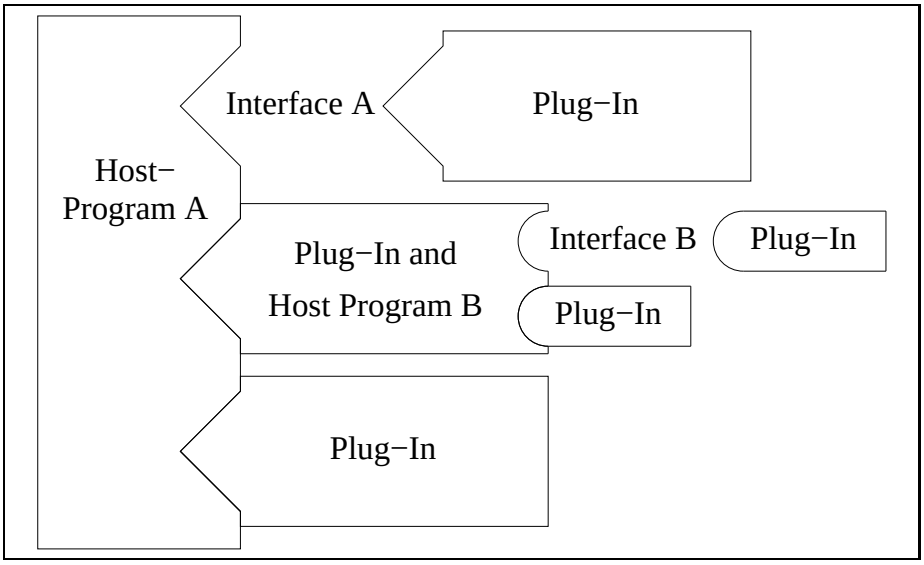

Figure 3.2. Hierarchical structures of plug-ins/host programs

The host program also provides the interface enabling plugging and unplugging of plug-ins. It is through this interface that a plug-in communicates with the host program (a description about interfaces is found in the next section). How a plug-in communicates with a host program depends on the interface and how it is provided by the host program. There is a possibility to create hierarchies of plug-ins in the same way as composite components can be built together in a hierarchical way, this is described in figure 3.2.

A host program (call it B) can, with a given interface, act as a plug-in to another host program (call it A). To enable this, B will need to have two interfaces, one allowing it to act as a host, and another to act as a plug-in. The interface between A and B does not have to be the same as the interface between B and its plug-ins.

3.4 Interface Between a Host Program and the Plug-ins

The *interface* that interconnects the host program with the plug-ins can be implemented in many ways and some of the possible plug-in interfaces will be described in this section. The main purpose of the interface is to enable plug-ins to be plugged and unplugged. The interface between a host program and the plug-ins is shown in figure 3.3. If the interface used in the host program does not match the interface in a plug-in, that plug-in will not be able to be plugged into the host program.

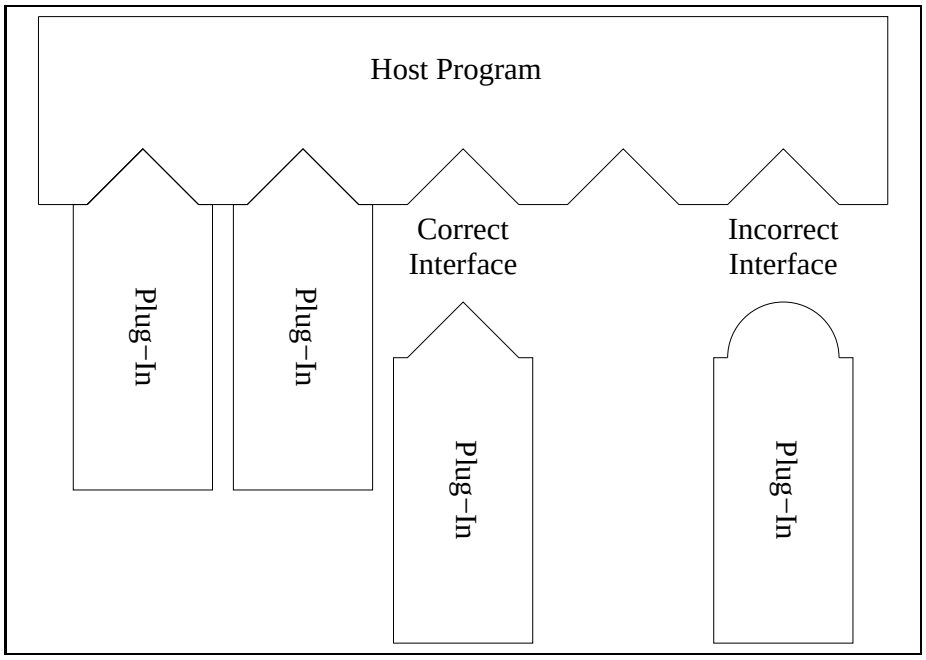

Figure 3.3. The interface between a host program and the plug-ins.

The interfaces described in this section are more or less common interfaces when a plug-in architecture is considered, there probably exist other interfaces that can be possible in a plug-in architecture but the ones covered here are considered to be the most relevant ones. The interfaces investigated are:

- Application Programming Interface (API),
- inheritance,
- virtual methods, abstract classes and object-interfaces,
- metadata,
- middleware, and
- protocols.

A general overview of the interface will be described, for each interface presented, as well as an application using the described interface (if one exists). Each section will be concluded with some requirements, advantages and drawbacks. Some of the interfaces presented below are dependent on specific functionality in the programming language used. Section 3.4.7 will describe under which circumstances a certain interface is appropriate.

3.4.1 Application Programming Interface

An *Application Programming Interface (API)* describes the expected indata, outdata and under what preconditions the interface can be used. The API is used by a developer when a software component is to be connected to another component. An API will often contain a library that the developer has to link with, tightly binding the developers program to the API [23].

An example of an API is the Java API. The API used in Java is not directly used for plug-ins but still, it presents some of the main ideas of an API, which also, applies to an API that can be used within a plug-in architecture. Using an API requires that the programmer knows how the API is defined or at least has the possibility to find documentation on the definition of the API. Each class in Java has its own API describing, for instance, how methods[2] and finals[3] are accessed by another object. The specification includes type definition and information on specific properties of that class [14].

If an API is used as the interface, then the following are requirements have to be satisfied:

- a host program must implement the subroutines, types and eventual call-order exactly as they are described in the API,
- to create a compatible plug-in the developer must follow the rules described by the API, a plug-in might be forced to implement a subroutine to make it pluggable/unpluggable, and
- the API cannot be updated without extreme caution, changes could lead to non-working plug-ins. New specifications that do not affect previously defined properties in the API is not a problem.

[2]"a group of programming statements that are given a name" [23].
[3]Used to describe a constant in Java [23]

Using an API to implement the interface between a host program and plug-ins will fulfill the requirement to be able to plug and unplug during runtime[4]. To plug a plug-in into a host program the host program calls a specific function which must be implemented in the plug-in. Another required function is called to unplug a plug-in. Beside these two fundamental subroutines a plug-in is controlled by the host program through the additional subroutines it specifies in the API [37, 28].

Netscape Communicator[5] uses an API to enable plug-ins and enhance the program with extra functionality. The plug-in API provided with Netscape was according to [28] designed to:

- "extend the capabilities of Communicator by providing inline viewers for types of data not supported by Communicator itself", and
- "provide an API that is as simple and concise as possible, making it relatively easy to leverage existing native code libraries or convert existing applications to take advantage of the web."

The plug-ins for Netscape Communicator must be written in C/C++. Existing programs can implement the required subroutines in the API making them plug-ins.

The services provided by Netscape plug-ins are mainly intended for use by Netscape Communicator and not by other plug-ins. A Netscape plug-in can provide the service to handle one or more MIME[6] types that the plug-in can decode and view. This means that the browser gains new functionality through the plug-in's ability to decode new MIME types. A Netscape plug-in can use resources defined in the API and provided by Netscape Communicator while operating [28].

An API will require the plug-in to implement a few subroutines to work as the host program expects. To have a working plug-in architecture implemented using an API requires the same programming language to be used in host program and plug-ins. A well documented API can make it easy for the developer of a plug-in to implement the correct subroutines and types. An API with a bad or non-existing documentation will require a lot of effort by the plug-in developer, resulting in fewer plug-ins or plug-ins with functions implemented in the "wrong way" compared to what was intended by the developer of the host program and the API. An API tends to invade all code layers and create a massive dependency between them, this would in one way ruin the concept of a plug-in as a separate component [33, 32].

It could be claimed that each of the interfaces in the following sections are APIs, and in some sense that would be true but they also provide a few specific techniques not described in the definition of an API.

[4]Requires that this is supported by the programming language

[5]A program used to browse web-pages, http://www.netscape.com

[6]Multipurpose Internet Mail Extensions, a specification for formatting non-ASCII messages so that they can be sent over the Internet[43].

3.4.2 Inheritance

When *inheritance* is considered within object oriented programming it refers to a class that inherits functionality and characteristics from another class known as the base class [23]. For example; a class that describes a car is inherited by another class which wishes to describe a sports car. The inheritance will give the class of the sports car the properties of the car class (which is the base class in this case), the other methods in the sports car class will be specific sports car methods. How this inheritance is implemented will vary depending on the programming language used, but the main concept of inheritance is often the same [41].

When the inheritance concept is considered as a technique to enable a plug-in architecture, a general plug-in class is constructed by the developer of the host program. The general plug-in class will be a base class and is inherited by all classes which wish to have the ability to be plugged into the host program as plug-ins. The functionality inherited from the base class by every plug-in is not anything that extends the host program, this is only to enable the plugging/unplugging. The functionality (in addition to this base functionality) is the one provided in the methods of the plug-ins that makes each plug-in unique.

An example of a host program that forces the plug-ins to inherit a plug-in base class is Estrutura, (which was created as a result of this research on plug-in interfaces). Estrutura is an object oriented system implemented in Perl using Tk[7] to implement the graphical user interface (GUI).

Estrutura is a general host program with no functionality except the one used to manage plug-ins. The following functionality is offered to a plug-in and can be used by the plug-in if the plug-in implements the subroutine that performs a certain function:

- ability to plug and unplug (enabled when the plug-in base class is inherited),
- manage services; a plug-in can offer services and request services from Estrutura,
- provide a frame where the plug-in can show a GUI,
- save and load data,
- the ability to use a cascade menu item in the host program's menubar, and
- show status information in the statusbar at the bottom of the Estrutura window.

Beside this functionality, Estrutura supports multiple workspaces where sets of plug-ins can be loaded. Plug-ins can be dependent on services provided

[7] An explanation of Tk is found in the glossary

by other plug-ins within the same workspace. The workspace functionality makes it unnecessary to open several instances of Estrutura. The workspace capability has nothing to do with the plug-in architecture and will not be further investigated here.

Inheriting a base class and the functionality of that class requires a few things from a plug-in. A plug-in must inherit the base class, and it must also call the constructor of the base class to instantiate the functionality in the base class. Some of the functionality offered by the host program might require that the plug-in implements the subroutines that are required for those functions. This requirement on functionality is in some ways like the requirements presented in the previous section on APIs, but the interface making the plugging and unplugging possible is not implemented through a classical API but through the inheritance of the base class.

The advantage of inheriting a base class is that the basic functionality of a plug-in is satisfied as soon as the class is inherited and the constructor of the base class is called.

The drawback of inheritance is that every plug-in must be written in the same programming language as the base class. Since the base class could be a separate pre-compiled class, depending on programming language, there might be a risk that the developer of a plug-in by mistake overloads some methods used for internal communication between the host program and the base class (could be avoided in eg. Java by using "final" methods). Inheritance require documentation on what the plug-in needs to fulfill in order to be pluggable.

3.4.3 Abstract Classes and Object-Interfaces

A special case of classes and methods that cannot be instantiated are *virtual methods* (C++), *abstract classes* (Java) and *object-interfaces*[8] (Java). These three language constructions affects the way a plug-in architecture can be implemented and will give the developers (both of the host program and of the plug-ins) new possibilities and restrictions.

An object-interface is described in Java as a collection of constants and abstract methods that does not have an implementation. The contents of a object-interface class is just a list of constants and method declaration with no body; only the method names and parameters are given. The methods specified in an object-interface class are called abstract methods and must be defined by all classes implementing the object-interface [23]. An object-interface will guarantee that each class implementing the object-interface will have some constants and methods with given input types and return types.

An abstract class can however contain both abstract and not abstract methods and data declarations other than constants. An abstract class can, just like an object-interface, not be instantiated since the class extending an abstract

[8]this interface will be called object-interface to separate it from the plug-in interface that uses interface throughout the rest of the report.

class must define the methods declared as abstract in the base class [23, 14].

C++ supports techniques much like the two described above. It is possible to declare virtual methods in C++ if they are declared as "virtual" and set to be equal to zero. Example declaration of a virtual method named meth: "virtual int meth()=0;" This will make the class abstract (to use a concept already described) and if all of the methods in a C++ class are virtual and no variable has been declared, the class could be called an object-interface [9].

There is no built in support for abstract classes or methods in Perl but the functionality of abstract classes are available as a Perl module [36]. The technique used in Perl is regular inheritance as presented in the previous section. Other programming languages supporting object orientation and possibly abstract classes and methods will not be presented here but could probably be used to implement a plug-in architecture in the way described below.

An existing system that uses an abstract base class which plug-ins inherits from is Eclipse. Except for a small kernel in Eclipse, all of the functionality is located in plug-ins [30]. Each plug-in can provide a number of extensions (known as extension points). The extensions can be used by other plug-ins or by the Eclipse platform. A plug-in usually consists of a small tool while a more complex tool often is split into several plug-ins. Eclipse provides an abstract base class which every plug-in class must extend. This base class provides configuration and management functions for the plug-ins and makes sure that the methods declared as abstract are implemented. Besides inheriting from a generic abstract plug-in class, Eclipse uses metadata to create a manifest for each plug-in, which contains information about the plug-in (more information on metadata are found in section 3.4.4).

One reason for using an abstract class as a base class is that all of the methods that are declared as abstract must be implemented in every plug-in. This requirement will make it easier for the developer of the host program to make assumptions on the plug-in. It will also relax the requirements for fault handling and runtime checks to see if methods are implemented. If an abstract class contains both abstract methods and regular methods, the host program can communicate with each plug-in through system services implemented in these regular methods and at the same time benefit from the advantages of abstract methods.

To use an interface or an abstract class, forcing the developer of a plug-in to implement all of the methods defined as abstract, could also be seen as an extra overhead. An interface or an abstract class is often one of the more advanced techniques within object oriented programming which might make it harder for developers with limited knowledge to implement a new plug-in.

3.4.4 Metadata

Metadata is data about data. Metadata can be thought of as library cards at a library, the cards contain information (data) about the literature (other data)

it refers to.

No program has been found during this investigation that only implements metadata as the interface between a host program and plug-ins. In the programs studied, metadata is used to describe what the host program needs to know to activate a plug-in. To create a plug-in interface by only using metadata would probably be possible and maybe preferable in some situations. How the metadata is passed or transported between plug-in and host program is not specified in the metadata, this is up to the developer of the host program.

One program that uses metadata as described above is Eclipse, the interface used to plug and unplug consists of the metadata information and the inheritance of an abstract class. XML (eXtensible Markup Language) is the format of the metadata used in Eclipse. It describes the manifest which is required by Eclipse from each plug-in. A plug-in's manifest contains information on version, name, identity and provider. Besides this obligatory minimal information, the manifest also contains information on dependencies and extensions. [7]

How an interface, only using metadata to implement the interface between a host program and the plug-ins, should be implemented is not clear since there is no program found that uses this technique. One requirement would be the ability to encode and decode the metadata used. The encoding/decoding of metadata must be performed in the same way at the host program and the plug-ins.

If metadata is used as the plug-in interface some of the following properties could be seen as advantages with metadata:

- different programming languages can be used in host program and plug-ins,
- the representation of metadata can be made in a common metadata language ie. XML,
- the properties described by the metadata can be extended with new labels and properties, and
- different computer architectures can be used, eg. alpha, mips, i386, etc.

The main argument for not using metadata as the interface is the encoding/decoding of metadata. This coding could slow down the communication between the host program and the plug-ins and, it will also require the developer of a plug-in to use the correct tags/labels when the metadata is encoded/decoded. To use metadata as the interface will require an extensive documentation on the encoding/decoding of data.

3.4.5 Middleware

Middleware is a concept within the research area of distributed systems. The purpose of a middleware is to isolate the application software from system soft-

ware, other underlying runtime services and their technical aspects [10].

Some of the advanced middleware solutions are Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) [8] and Simple Object Access Protocol (SOAP) [38]. Since a middleware is used to support a distributed system the concerns of where the plug-ins are located will be less important.

One solution that uses CORBA to implement a plug-in based architecture is the UMTS network element provisioning tool. The purpose of the tool is to provide the UMTS networks operators with a tool that makes it easier to configure different aspects of the UTRAN [1, 3].

The technical documentation of UMTS network element provisioning tool does not include a description of the plug-in architecture, it only contains a brief overview of the main functionality. To describe this system further would not provide any new insights to the use of a middleware as a plug-in interface. The main thing to learn from the UMTS tool is that it is possible to implement a plug-in architecture with a middleware as the interface between the host program and the plug-ins.

One possible design of a system that uses a middleware as the plug-in interface would be to have the host program located at the user (as a client application) and the plug-ins located at some remote servers. This solution would be supported by many of the middlewares that uses a client/server model. The plug-in interface would be specified in the middleware and be similar to a regular API from the plug-in developer's point of view.

To use a middleware as the interface will provide the ability to have distributed plug-ins which could be spread over different servers at different locations. Depending on the middleware used to implement the plug-in interface, the programming language used in the host program and the plug-ins might not be the same (eg. CORBA supports different programming languages while Java RMI must be used together with JNI (Java Native Interface) to support other languages). Access to distributed plug-ins can be achieved in several ways: over the Internet, on the local network or on the same machine. Exactly how plug-ins are distributed and made available to the user will depend on the design of the host program.

Distributed systems are known to have a high latency. Loading a plug-in which is located at a remote server will probably be slower than to load it from the same computer as the host program is running on. The middleware defines the interface making it more or less complex, this could lead to difficulties in implementing plug-ins compatible with the middleware.

3.4.6 Protocol

The concept of *protocols* is well used within computer networking. In fact, the Internet as we know it today consists of protocols controlling almost everything.

In this section the following definition of a protocol will be used:

> "A protocol defines the format and the order of messages exchanged between two or more communicating entities, as well as the actions taken on the transmission and/or receipt of a message or other event." [22]

A protocol can be considered to be very much like an API, the similarities are that a protocol and an API defines the rules for the communication between two or more entities. Netscape refers to the communication between a plug-in and the API as a protocol[28] even though it is a classic API given the definition used in this thesis. The differences between an API and a protocol are that an API requires linking to a library while a protocol is standalone and only defines the request/response order and sometimes a common way of transporting the communication.

If a protocol is used to define the plug-in interface, an underlying layer is required to transport the protocol messages. This transport layer is either an existing transport protocol or a new one specified by the developer of the protocol. Using a protocol to plug and unplug will require that these two actions are defined by the protocol. If the host program would like to add more functionality then additional messages needs to be defined or extra parameters can be added to the definition of an already existing message. To use a protocol as the plug-in interface will probably lead to a complex specification where encoding and decoding of data will need careful consideration [22, 47]. One language used to define protocols are ASN.1 which could make the definition of a protocol easier [12].

No program found during this investigation uses a protocol to exchange messages between a host program and the plug-ins. To use a protocol as an interface would require, as mentioned earlier, a transport layer for the protocol. How the messages between host program and plug-ins are transported is not as interesting (in this context) as how the messages are defined. Building a protocol can be very complex and require that the designer is very foreseeing on the future use of the host program, possible plug-ins, and their functionality. An example of a protocol stuck with an old definition limiting applications and new techniques is the Internet Protocol version 4 (IPv4). It will take time before existing applications are adopted to the new protocol even though a new protocol[9] has been designed. The documentation of a protocol is as important for a plug-in developer as it is when an API is used [6].

Using a protocol will enable the plug-ins to reside at any server as long as it is connected to the client where the user is running the host program. A protocol will not require the programming language to be the same in host program and plug-ins. It is enough if the encoding and decoding of data is made in the same way. One advantage compared to the API concept is that a protocol will enforce a layered structure of the architecture since it does not require a library to be included. [22].

[9] Internet Protocol version 6 (IPv6)

A protocol will often suffer from a complex and hard to adjust definition. To alter an existing protocol often requires updates of every implementation of that protocol.

If a plug-in is located far away from the host program, then the round trip time will cause a delay of the response from the plug-in which could be a problem if performance is an important issue [22].

3.4.7 Discussion

The decision of which interface should be used under certain preconditions is not easy. The interfaces described in the previous sections presents some possible solutions, each with certain requirements, advantages and drawbacks.

An interface used in a plug-in architecture has, as it has been presented here, one main purpose: to support plugging and unplugging of plug-ins to/from the host program. The functions provided beside this main functionality could be considered as extra features that might handle services, dependencies, resources, authentication, coordination etc. one interface technique might be preferred to another depending on the extra functionality needed.

Some factors to consider before a plug-in interface is chosen:

- main functionality of the host program, to host plug-ins or to provide functionality that the plug-ins can extend or alter?
- are the developers of the plug-ins the same people as the vendor of the host program or ordinary users/developers?
- is the plug-in architecture added to an existing program or created from scratch? What previous functionality might exist?
- will plug-ins run locally or on a remote server?
- how will the extension offered by a plug-in be provided? By services, events, overloading or by other means,
- security of the host program, will there be a risk that malicious or untested plug-ins are developed?
- end users, who will use the system, how easy should it be to load a plug-in and to download/buy a new one?
- dependencies, can a plug-in or the host program depend on another plug-in or service?
- resources and performance, is it a real time system with limits on memory usage, CPU-time or overall performance?
- coordination, will the plug-ins have to be coordinated in some way?
- stability, how reliable does the host program have to be, is a plug-in allowed to cause the host program to crash?

Host program

The host program might be an existing program that is to be updated with a plug-in interface or a new program where the plug-in interface is integrated in the planning and requirement specification. Which of the interfaces to use may be dependent on the programming language to be used in the host program. If the language does not support object orientation, then inheritance, virtual methods, abstract classes and object-interfaces can not be used (some of the object oriented middlewares will also not be possible).

An existing program which is upgraded with the possibility for plug-ins will probably limit the number of interfaces that could be used, depending on the programming language and of the architecture used in the existing program.

Plug-in developers

The choice of interface will vary depending on who the proposed developer of plug-ins is. If the plug-ins are developed by the same developer who has created the host program then the insight in the host program is complete making it easier to integrate the plug-ins with the host program. If the intention is that a developer other than the one who created the host program should be able to write plug-ins, then documentation and probably an example plug-in which clearly demonstrates how the interface works is needed to simplify the implementation of a new plug-in.

The interface chosen depending on the developer has much to do with how well documented the interface is, since this is related to how easy it is to implement an interface. If an interface is used where plug-ins can be developed in any programming language, then there will of course be more potential developers. A well documented API, abstract class, virtual methods or object-interface will probably not be any problem for an experienced developer.

Distribution

The method used for the distribution of plug-ins will be directly connected to who develops the plug-ins. If the producer of the host program creates the plug-ins, the plug-ins will probably be distributed on the producer's webpage or through an update system built into the host program. If the plug-ins are developed by the users then public archives with the ability to download the plug-ins are needed if the plug-ins are to spread among the users.

Services

If a plug-in should be able to provide a service to the host program or to another plug-in then the interface needs to support this additional feature. The

interfaces presented previously all have the ability to add extra services, the difference between the interfaces is the amount of effort a developer of an interface has to put in to support new services.

Sharing data and resources between plug-ins will be easier if the interface used, uses the same programming language in the host program and the plug-ins. The reason for this is that serialization and deserialization of data would be needed if an interface is used where the data is converted to fit a protocol, metadata or some other general data structure.

Dependencies

A requirement to allow a plug-in to be dependent of a certain service or other plug-in is that the interface has the ability to tell a plug-in if a service is provided and by whom or where. How this messaging is done will depend on the interface used, eg. if a middleware is used a plug-in might get the IP-address to the provider of a requested service. If inheritance is used instead, a pointer to the plug-in-object providing the service might be passed instead. Whether a plug-in gets access to another plug-in or only to the requested service is up to the developer of the host program, and the decision will not affect the choice of interface.

Local or remote plug-ins

An important consideration is whether the plug-ins will execute on the same machine as the host program or if they are to be executed on a remote server. To support remote plug-ins either middleware, metadata or a protocol could be used. This will require that the data transported is serialized and deserialized in some way. For plug-ins executing locally, any of the interfaces can be used even if the one that can be implemented in the same programming language will have a natural advantage in performance.

Security

Security is something that could be of importance to mention, even if this is a very complex area which probably needs a separate investigation. Implementation of security in a plug-in interface is not always needed. Implementing authentication, certificates or password will often come with an overhead.

If a host program is used within a sensitive or high security environment like a bank, the military or a nuclear plant, then the need for security and testing is massive. A host program might require that a plug-in is certified during the authentication before it is plugged into the host program. To require a secure interface could include that the communication between a plug-in and the host program is encrypted, that they share a symmetric key, or that a public key of a plug-in is matched with a private key included in the host program. To chose

an interface depending on the security requirement will depend on if a plug-in is executed locally or remote. Any of the suggested interfaces could be used and which one to choose should be dependent on the security techniques available for that specific interface [4].

User

Who the user of the host program is can affect the choice of the plug-in interface. The plug-ins might be loaded by a system administrator in a terminal window, using a command line interface to supply parameters. The opposite would be a user selecting a plug-in on a webpage that automatically downloads and installs the plug-in to a plug-in directory, ready to be plugged in by the host program. A user interface which is easy to use will require the interface to support debug- and install functionality. The requirements on the interface will be the same here as if they would provide any additional services.

Resources and performance

The requirement on a plug-in can vary from almost nothing to very strict resource limits considering memory usage and CPU-time. The highest performing interface will be one that uses the same language in host program and plug-ins. To use a middleware is not preferred if there are realtime requirements and memory limitations. The interface that can be kept smallest and with minimum memory usage is the API-interface, this will also, since it is written in the same programming language, be fast. An object oriented interface will also work fine but will probably require more memory than an API.

Coordination

Coordination of plug-ins could be needed if they are allowed to execute simultaneous and may use shared memory. In many common operating systems this is solved with semaphores or signaling. If coordination is needed between the plug-ins and host program any of the interfaces could be chosen. An easy solution to provide coordination could be to implement it as a service provided by the host program or as a coordination-plug-in.

Stability

If a host program is required to run without crashes or unwanted impact of the plug-ins, the plug-in interface could be of importance. If an interface is chosen which executes the plug-ins in the same process as the host program there is of course a risk that a plug-in might crash the host program if the plug-in crashes. If an interface is chosen where messages are passed in some way then the risk

is not as high since the host program can choose only to process messages that are in a correct format.

3.5 Existing Solutions

There exists several programs that implement some sort of plug-in architecture. It is remarkable that there exists so many products on the market that use a plug-in architecture even if the research on the specific area of plug-ins to this date is very sparse. The research on subjects related to plug-ins are presented in section 3.6.

Some of the existing software that exists today and that have been investigated during the research on plug-in interfaces are:

- Eclipse [30],
- Netscape [28],
- Photoshop [2], and
- UMTS provisioning tool [3],

all implementing some of the plug-in interfaces described above.

There exists programs in almost every genre of software that uses a plug-in architecture, from large scale server programs [29] to small programs running on a mobile phone [31]. An observed tendency for new programs or new versions of existing ones is that the number of programs that implements a plug-in architecture increases[10].

3.6 Related Work

There are some research areas either directly related to plug-ins or in the way that they try to find solutions to the same problems in another way.

Aspect oriented programming is in some ways concerned with dynamic platforms and interfaces, the research concerns how aspect oriented components can be composed dynamically during runtime [34]. The composition of components is in a way like plug-ins, a plug-in could be seen as a component but the aspect oriented programming is focused on a new way of programming as a response to object oriented programming.

The component oriented programming concept presents some suggestions on pervasive software that will benefit from increased reuse, reliability, and

[10] this is personal observation not based on any scientific results

efficiency [13]. Plug-ins may benefit from the work on component-oriented programming since the research on components indirectly affect how plug-ins are developed (a plug-in could be seen as a component).

The research on dynamic software architectures study dynamic software during runtime. This means that the composition of components are done during runtime [40, 25]. Compared to plug-ins the research on dynamic software is trying to solve one of the problems that plug-ins also focuses on but from a different angle. Software that can modify itself during runtime is achieved by plug-ins, they can be plugged/unplugged, this is a bit different approach than dynamic software investigates though.

There are probably more research areas related to plug-ins than the ones presented above. The ones presented here have been found during the search for information on plug-ins.

Chapter 4

Problem Specification

The problem this thesis is trying to solve originates, as mentioned earlier, from TietoEnator in Umeå. Some of the departments at TietoEnator are working with software development and, during the development, execution of the developed software and analysis of the log-files from these executions are needed. The developers either execute the software (called load modules) in terminal windows or with some smaller unofficial program developed by themselves and, analysis, of the logfiles, is carried out in the same unstructured way. This way of executing load modules and analyzing log-files has led to a number of, more or less, sophisticated applications where load modules can be executed and log-files analyzed.

To solve the problem described above, a tool able to execute load modules and analyze trace logs was needed. If this tool was to be used it had to:

- supply desired functionality and services,
- be integrated into the existing environment, and
- have a low threshold for learning but still be powerful.

To achieve the requirements for these requests it was suggested by TietoEnator that a tool or a framework that is dynamic and extensible was to be developed. Exactly which functions and characteristics this tool was going to have was not specified. One solution was to find an existing tool, alternatively develop a new one.

The programming language used within almost every smaller tool at TietoEnator in Umeå is Perl and, therefore it would be good, but not required, that the tool was developed in Perl. This would make it extensible, maintainable and enable further development within TietoEnator.

The following chapters will explain how the development of this tool has been carried out. In chapter 6 the results of the study will be presented as a technical description of the tool.

Chapter 5

Implementation

This chapter will describe how the study was carried out and which important steps and decisions made. Detailed information on the progress of the work can be found on the thesis web page[1].

The study started with an introduction to how the work would be carried out and organized at TietoEnator. To understand how the solution to the problem was to fit into the organization, it was important to understand the structure of the organization, and how existing tools were used.

As a first step, a specification of the thesis work was made. In the specification a project plan was provided together with the background information of the problem from TietoEnator's point of view (this information was presented in chapter 2). The project plan was a good way to divide the work into stages to get an overview of the project, the plan was general and intended to be updated during the progress of the work.

The requirements on the tool were important to be able to start the investigation of existing tools, research on techniques and methods to implement in a new tool.

5.1 Analysis and Description

The first step toward a detailed specification of the tool was to write down some general requirements that was gathered during the investigation of the organization and how the tool was to be used. It was suggested by TietoEnator that the tool should not contain any functionality except the ability to use new tools together in a way configured at runtime. The tool should have the ability to support plug-ins and work as a host program for other tools, the development of a new load module execution/trace and error analyzer tool/plug-in was not a primary goal. The future plug-ins that should be supported could contain any functionality, plug-ins with the following functionality were considered as possible plug-ins: analysis of trace logs, execution of load modules and generation of state diagrams.

The tool is called the host program since it will support a plug-in architec-

[1] http://www.cs.umu.se/~c00stm

ture. The following general requirements on the host program were presented to TietoEnator:

- initialize plug-ins during runtime (plug/unplug plug-ins),
- offer a GUI,
- the host program should be written in Perl,
- have an interface which makes it easy to write new plug-ins,
- ability to save/load settings and states,
- an archive of plug-ins where plug-ins could be gathered,
- work in the existing development infrastructure, and
- object oriented structure, to support exchangeable modules.

The requirements above were discussed in order to redefine some requirements and to suggest some new requirements. After a review of the requirements by TietoEnator the following additional requirements were suggested:

- more than one instance of the same plug-in should be able to be loaded at the same time, with some exceptions eg. singleton-plug-ins and plug-ins that affect the host program on the highest level,
- support a plug-in "sand-box", a workspace, where plug-ins can be loaded,
- possibility to have several (at least ten) workspaces, running at the same time without any influence of the workspace on each other, and
- ability to debug the host program and the plug-ins:
  - trace printouts of plug-in execution,
  - ability to debug separate instances of plug-ins, and
  - debug the source code of the plug-ins, ie. with Perl's built in debugger.

5.2 Specification and Categorization

After the analysis and the first description, a specification of the host program was written that presented the functional and non-functional requirements. The requirements below include the requirements from the review, and the comments after the first general requirements.

The specification was not supposed to be a complete specification with a complete requirement list, instead, it was supposed to work as a guideline where the solution was to be reviewed and evaluated by TietoEnator continuously during the development.

The following specification is the one used when the theoretical study was carried out and later on when the prototype was developed. The priority of requirements was decided together with TietoEnator.

5.2.1 Requirements that absolutely must be met

Functional requirements:

- initialize plug-ins during runtime (plug/unplug plug-ins),
- dependencies, plug-ins should be able to use functionality provided by other plug-ins, and
- plug-ins must be able to show graphical information in the host programs GUI.

Non-functional requirements:

- offer a GUI, and
- work in the existing development infrastructure.

5.2.2 Highly desirable but not necessary requirements

Functional requirements:

- ability to save/load settings and states, for the host program and the plug-ins,
- more than one instance of the same plug-in should be able to be loaded at the same time, with some exceptions eg. singleton-plug-ins and plug-ins that affects the host program on the highest level,
- support a plug-in "sand-box", a workspace, where plug-ins can be loaded,
- workspaces should not affect each other,
- a plug-in should be able to print debug information during execution, and
- a plug-in should have the ability to choose if it would like to use a GUI or not.

Non-functional requirements:

- possibility to have several (at least ten) workspaces, started at the same time.

5.2.3 Requirements that are possible but could be eliminated

Functional requirements:

- update the list of plug-ins during execution, and
- have the ability to debug separate instances of a plug-in.

Non-functional requirements:

- an archive, database or webpage where plug-ins could be gathered,
- object oriented structure, to support exchangeable modules. and
- debug the source code of the plug-ins, eg. with Perl's built in debugger.

5.3 Prototyping and Evaluation

After the specification of the requirements, the theoretical study was carried out, investigating existing plug-in systems, specifically the interface between a host program and plug-ins. After this theoretical study the best solution was to develop a new host program able to fulfill the specified requirements. The reason why Eclipse or other existing host program where not chosen as the plug-in system was much because it was a great advantage to be able to write the plug-ins in Perl. None of the existing host programs that fulfilled the other requirements allowed this.

The first prototype of the GUI was a coarse sketch; infact the intention was that this sketch should be rough and invite to suggestions on change and new features. After evaluation by TietoEnator, some changes were suggested considering the design of the next prototype. The next prototype was an upgraded design, a computer made sketch, The purpose of this prototype was to give a preview of how the GUI would look like in the program to be created, this GUI prototype can be found in appendix A. After evaluation, some minor changes were suggested, to be considered in the development of the final program.

When the decision on how the GUI should look like was made, there was still one main issue: which interface was to be used to connect the plug-ins to the host program? The theoretical study on plug-in interfaces proceeded with the results presented in chapter 3. When the theoretical study was completed it was decided that inheritance was the best plug-in interface and should be used for the plug-in architecture. Inheritance was chosen since the programming language Perl supports object orientation. Inheritance would also make it easy for a developer of new plug-ins to make them work with the host program. One drawback of inheritance is that the same language must be used, but a solution to how this obstacle could be overcomed in the future was proposed. If a glue

layer between the host program and a plug-in is used, then it could be possible to use different programming languages, with the glue layer as an interpreter.

5.4 Implementation and Presentation

It was specified that Perl should be used as the programming language (since this was the language used for tools) to implement the host program. It was decided that Tk should be used to create the GUI. Tk was chosen because it has a Perl module, also since Tk has been previously used at TietoEnator together with Perl. Perl supports object orientation which was one of the decisive factors when inheritance was chosen for the plug-in interface.

A design of the main components of the host program was created before the implementation began. The design of the system had besides the plug-in interface also the workspaces to consider as a special part of the system. Before the implementation started, some study was needed on Perl/Tk [24] and Perl's support for object orientation [41].

The implementation of the host program started according to the schedule. It was easy to get quick answers regarding design issues during the implementation since the work was carried out at TietoEnator. The host program was named Estrutura which means framework in Portuguese, the name has no other specific meaning related to the research area.

A 1/3-presentation[2] was held when the main functionality had been implemented. The main focus of the presentation was the system design of Estrutura and its plug-in architecture, but the general progress of the thesis work was also presented. On this presentation there were many important questions and change requests regarding functionality and future usage. The suggestions from the presentation were considered and some of them resulted in changes to Estrutura.

Estrutura has, beside the original specification, been enhanced with some extra functionality, both with respect to the plug-ins but also in the operability and user interface of Estrutura itself.

[2]the progress of the thesis project was 1/3 of the total progress.

Chapter 6

Results

In this chapter, Estrutura and it's plug-in interface, which is the result of the investigation, will be presented as a solution to the problems described in chapter 4. This chapter will give a technical and partly a functional, overview of Estrutura. The results of the plug-in interface investigation were presented in chapter 3.

6.1 Systems Description

A high level overview of Estrutura is illustrated in figure 6.1. The plug-in base class is not considered to be a part of the host program even if it is developed together with the host program to implement the plug-in interface.

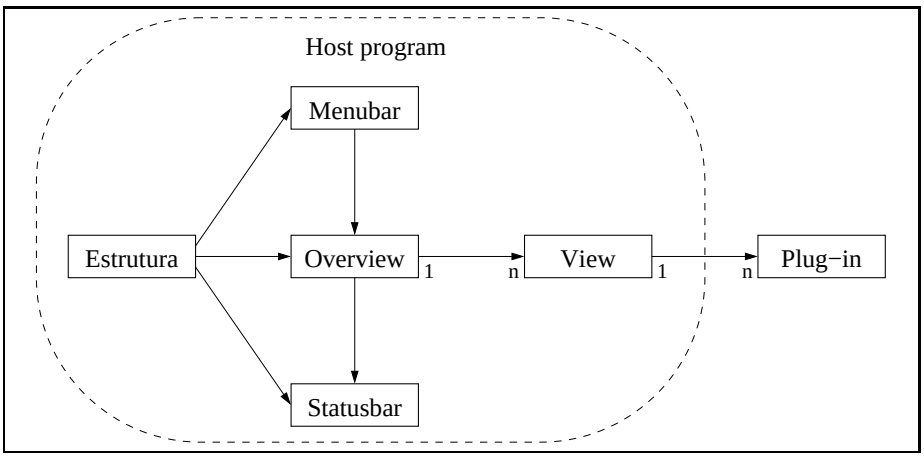

Figure 6.1. Estrutura - system overview.

6.1.1 System overview

Estrutura consists of one main package which is used to coordinate the start sequence of the GUI, this main package is named Estrutura. The GUI is divided into three different packages:

- menu - contains the menubar on the top of the GUI,
- workspaceManager - the frame where the list of workspaces and the currently active workspace is shown, and
- statusbar - a frame in the bottom of the GUI where status information can be shown.

The interaction between these three packages are implemented through subroutine calls and passing of references. The arrows in figure 6.1 on the facing page indicates which package that is that is depending on another package's subroutines.

The source code of Estrutura and some example plug-ins can be found on the thesis web page at: `http://www.cs.umu.se/~c00stm`

Estrutura

The main package Estrutura does not consist of any subroutines. The only functionality provided in Estrutura is calls to the other packages, which starts the system in a correct way. The GUI implemented by Tk is started in Estrutura and references to the MainWindow are passed to the menu, workspaceManager and statusbar, which initializes the graphical content.

Menu

The menu consists of a frame where menu buttons with cascading menu items are shown. The subroutines within the menu package are used to initialize the menu and to update the menu with the correct content during runtime. The following subroutines are a part of the menu package:

addPlugins Purpose: Search the plug-in directories for plug-ins, used by update plugPluginMenu to refresh the available plug-ins.
Indata: A reference to the menu to update and a path to search for plug-ins.
Outdata: The plug-ins found in the sought path.

fileMenuItems Purpose: Returns the menu items contained in the File menu button
Outdata: An anonymous array with the File menu items.

helpMenuItems Purpose: Returns the menu items contained in the Help menu button
Outdata: An anonymous array with the Help menu items.

openWorkspace Purpose: Show a dialog of which workspace to load, calls workspaceManager's openWorkspace with the path to the saved workspace as parameter.

pluginMenuItems Purpose: Returns the menu items contained in the Plug-ins menu button
Outdata: An anonymous array with the Plug-ins menu items.

showMenu Purpose: Initializes the menu when Estrutura starts, only called one time
Indata: a reference to the top level window and the path to the plug-in directories.
Outdata: a reference to the menu button which can be used by an active plug-in.

updateClosePlugin Purpose: Update the "Close Plug-in" menu items with the plug-ins currently loaded in the active workspace.

updateLoadedPlugins Purpose: Update the "Loaded Plug-ins" menu items with the plug-ins currently loaded in the active workspace.

updatePluginMenu Purpose: Updates the list of loadable plug-ins.
Indata: A reference to the menu to update.

workspaceManager

The workspaceManager package controls the workspaces. Every workspace is a separate object which is initialized by workspaceManager. The control of which workspace to show, remove or create is managed from this package. The subroutines included in the workspaceManager package are the following:

addWorkspace Purpose: Used to create a new workspace, initialize a workspace-object and create a notebook for the workspace.
Indata: The name for the new workspace.

workspaceButtonMenu Purpose: Show a workspace-menu if the right mouse button is pressed on a workspace button.
Indata: Which mouse button that is pressed and a reference to the workspace that the workspace button refers to.

activatePlugin Purpose: Set an already loaded plug-in as the active plug-in.
Indata: The identity number of the plug-in to set as active.

changeActiveWorkspace Purpose: Change the currently active workspace.
Indata: A workspace to set as the new active workspace.

closePlugin Purpose: Close a loaded plug-in.
Indata: The identity number of the plug-in to close.

getActivePlugin Purpose: Return a reference to the active plug-in in the active workspace.
Outdata: A reference to the active plug-in.

getLoadedPlugins Purpose: Return a list of the loaded plug-ins in the active workspace, calls getLoadedPlugins in the active workspace.
Outdata: A list of loaded plug-ins in the active workspace.

openPlugin Purpose: Open a not yet loaded plug-in in the active workspace.
Indata: A path to the plug-in to load.

openWorkspace Purpose: Open a previously saved workspace state from a file.
Indata: The path to the saved data.

removeWorkspace Purpose: Removes a workspace from the program.
Indata: A reference to the workspace to remove.

renameWorkspaceDialog Purpose: Show a dialog to change name for a workspace.
Indata: A reference to change name for.

saveWorkspaceDialog Purpose: Show a dialog to save the state of a workspace, calls saveWorkspaceState to save the workspace.
Indata: A widget to use as the parent of the dialog and a reference to the workspace to be saved.

saveWorkspaceState Purpose: Save the state of a workspace and possibly its plug-ins to a file.
Indata: The filename to save the workspace as and a reference to the workspace that will be saved.

showWorkspace Purpose: Initializes the workspaceManager GUI, shows the frame with the workspace buttons and the frame where the active workspace is shown.
Indata: A reference to the MainWindow and a reference to the plug-in menu.

Statusbar

The statusbar is implemented by using a frame where the content can be shown. The purpose of the statusbar is to show information about the workspace or the currently active plug-in in a statusbar frame at the bottom of the GUI. The statusbar package only contains these two subroutines:

getStatusbar Purpose: Return a reference to the statusbar frame. Outdata: A reference to the statusbar frame.

showStatusbar Purpose: Show the statusbar when the program starts, this subroutine is only called one time by the Estrutura package.
Indata: A reference to the MainWindow, used to initialize other graphical content.

6.1.2 Workspace

The workspace package/class is used when workspace objects are instantiated. A workspace object consists of a workspace button which is used to activate the workspace, and a notebook which is used to show graphical content of loaded plug-ins. The workspace class consists of the following subroutines/methods:

addServices Purpose: Register the services provided by a plug-in as available to the other plug-ins in the workspace.
Indata: a reference to the plug-in providing the services.

changeActivePlugin Purpose: Change the currently active plug-in.
Indata: The plug-in to set as the active plug-in.

closePlugin Purpose: Close a loaded plug-in.
Indata: A reference to the plug-in to close.

dependencyCheck Purpose: Find out if all of the dependencies of a plug-in are satisfied.
Indata: The plug-in to check the dependencies for.
Outdata: A list of the non satisfied dependencies.

getActivePlugin Purpose: Return the currently active plug-in in the workspace.
Outdata: The loaded plug-in.

getLoadedPlugins Purpose: Return all of the loaded plug-ins in the workspace.
Outdata: A hash with all of the plug-ins.

getName Purpose: Return the name of the workspace.
Outdata: The name of the workspace.

getServiceProvider Purpose: Return the plug-ins that provides a requested service, show a dialog where the user can choose provider if more than one provider exists.
Indata: The name of the requested service.
Outdata: The unique identity of the plug-in that provides the service.

hide Purpose: Called by the workspaceManager package to hide a workspace.

loadState Purpose: Load a saved state for a workspace and possibly the plug-ins loaded in that workspace.
Indata: The data to be restored to a workspace.

new Purpose: The constructor of the workspace-object, used to initialize a new workspace.
Indata: A reference to a notebook, a reference to a button, the name of the workspace and a reference to the menu button that can be used by plug-ins.
Outdata: Return a reference to the newly created workspace object.

openPlugin Purpose: Load a plug-in and register the services provided by it.
Indata: The path to the plug-in to be loaded.
Outdata: A reference to the plug-in-object.

remove Purpose: Remove the workspace.

removeServices Purpose: Remove all services provided by a given plug-in.
Indata: The plug-in to remove the services for.

saveState Purpose: Save the state of the workspace, the information that will be saved are information of: loaded plug-ins, identity of those plug-ins, offered services and the currently used service providers.
Outdata: The data to be saved.

setName Purpose: Change the name for the workspace.
Indata: The new name for the workspace.
Outdata: The new name of the workspace.

show Purpose: Called by the workspaceManager package to make a workspace visible.

6.1.3 Plug-in base class

The plug-in base class only consists of basic functionality used by the workspace to implement the interface. The following subroutines are implemented in the plug-in base class:

getDependencies Purpose: Return a list of which services this plug-in are dependent on.
Outdata: A list of dependencies.

getFileName Purpose: Return the filename of the plug-in.
Outdata: The filename of the plug-in.

getID Purpose: Return the unique number identifying the plug-in.
Outdata: The unique number.

getInformation Purpose: Return information of the plug-in.
Outdata: A hash with information of the plug-in.

getServices Purpose: Return a list of the services provided by this plug-in.
Outdata: A list of services.

plug Purpose: This is the subroutine used as a constructor to all of the data to be used in the plug-in. This subroutine must be called by the plug-in inheriting this base class.
Indata: The unique identity of the plug-in and the filename of the plug-in.
Outdata: A reference to the plug-in-object created.

setServiceProvider Purpose: Provide the plug-in with a reference to another object that provides a requested service.
Indata: The service provided and a reference to the object providing the service.

setStatusbar Purpose: Provide the plug-in with a frame that can be used as a statusbar when the plug-in is active.
Indata: A reference to the statusbar.

6.2 Plug-in interface

The plug-in interface is created via inheritance, if a new plug-in is to be created to Estrutura it must inherit from the plug-in base class and call it's constructor. The interface does not specify a special function for unplugging, this is taken care of by the garbage collector when a plug-in is dereferenced. The services provided to a plug-in through the interface are the following:

- ability to be plugged/unplugged,
- provide services,
- depend on services,
- save data,
- load data,
- show graphical content in Estrutura's GUI,
- provide a menu that can be shown in Estrutura's menubar, and
- show status information in Estrutura's statusbar.

To implement the additional functionality a plug-in has to implement a method with a predefined name. Eg. if a plug-in would like to show a GUI it must implement a method called "showGUI". The other functionality is also provided by the use of methods with predefined names.

The plug-in base class contains these optional functions as comments in the end of the source code file where the expected indata and outdata also are described.

6.3 Algorithm Description

The implementation of Estrutura is made with algorithms that are well documented in the code with comments and explanations. The only algorithm of direct interest is the algorithm used to plug a plug-in into Estrutura.

The algorithm used for plugging is located in the workspace package and consists of the following steps:

1 Perform a basic test, to see if the plug-in have the methods getID, getFileName and getInformation, do not continue the plugging if this test fails.

2 Create a unique number consisting of the number of milliseconds since 1970.

3 Call the plug method in the plug-in to be loaded.

4 Try to find services that the plug-in is dependent on, notify the user if any of the services is missing.

5 For the services that has a provider, set each provider for the requested service in the plug-in.

6 Register the services provided by the plug-in.

7 If the plug-in contains the method "run":

  7.1 If the plug-in contains the method "showGUI":

    7.1.1 Create a new page in the workspace's notebook.

    7.1.2 Create a new scrollable Pane to show in the page.

    7.1.3 Call the plug-ins "showGUI" method with the scrollable pane as the parameter.

    7.1.4 Raise the created page in the notebook.

  7.2 Change the active plug-in to the one plugged.

  7.3 Call the plug-ins run method.

8 Return a reference to the plug-in object.

Chapter 7

Conclusions

The thesis work has consisted of two parts. One more theoretical part with the investigation of the plug-in interface as the main focus, and another more practical part where the theories have been applied and implemented as a solution to an actual problem.

7.1 Plug-in Interface

To designate an interface that is outstanding, always working and best in every situation is not possible. The research on interfaces rather suggests guidelines and properties that needs to be considered when a plug-in interface is chosen. These guidelines can hopefully help a software developer/designer to avoid some common mistakes and design a system that has plug-ins as a natural part of the system.

One important question is whether this research has contributed with something new to the genre of software engineering. Even if the techniques of interfaces are not new, the usage of the interfaces and some of the aspects considered on these interfaces are proposed to be new. The discussion on which interface to use and under which circumstances, contain some new thoughts and reflections.

7.2 Problems and Reflections

This section contains my personal reflection on the thesis project. The thesis project has been on schedule the whole time. This is a bit remarkable considered the fact that I didn't know exactly what I was going to do when I first made the general schedule.

In the beginning of the investigation the hardest and most demanding thing was to understand what the thesis should investigate. The second most demanding issue was how the work should fit in to TietoEnator's organization and aid their software development. Both of these issues took some time to sort out but got clearer as the work proceeded.

Since the work was carried out at TietoEnator it was great to get quick

answers to some details on Estrutura or to get a second opinion on some other question.

The work has been interesting and made several issues clearer which where not considered before, regarding software engineering and plug-in architectures. I believe that user customization of software which is provided when plug-ins are used, gives a program characteristics both positive and negative. The positive thing is that every user can configure the program to work as he/she wants. One negative aspect is that it will be a lot harder to provide support for 3rd party developed plug-ins, since the developer of a plug-in has to provide this.

The plug-ins as a research area will hopefully be further investigated or be replaced by another more sophisticated technique.

7.3 Limitations and Future Improvements of Estrutura

There are a few functions that would be of interest to have as a part of Estrutura that is not yet implemented. Some of them are:

- a personal configuration file where settings can be saved,
- ability to change the provider of a service during runtime,
- right-click with the mouse on the plug-in-tabs to get a plug-in menu,
- add a menu item to the help menu where information of every loadable plug-in can be presented,
- support trace print-outs of calls to plug-in methods,
- version management for services, eg. dependency of a service of version higher than X.X, and
- ability to autoload plug-ins that provides a service that is requested but not yet loaded.

There is one requirement from the original specification yet not implemented. The not yet implemented requirement is "a plug-in should be able to print debug information during execution". The main idea with this requirement is that Estrutura can show debug-information of the method calls made through the plug-in interface and between plug-ins. As the implementation is today, each plug-in is responsible for printing of debug information.

Estrutura executes in a Unix environment, it has not been tested on any other architecture or operating system. Estrutura should work on a system where Perl is available together with the graphical library Tk.

One limitation on the ability to load plug-ins is that the plug-ins must be object oriented to be able to open more than *one* instance of that plug-in.

The future use of Estrutura will probably be as a part of the tools used today at TietoEnator. The ability to share services between plug-ins makes it easy to build structures of plug-ins that together form a new service or solution to a problem.

Chapter 8

Acknowledgments

This thesis would not have been possible without the help from the instructors and TietoEnator.

I would like to thank my instructor Gunnar Strand at TietoEnator for his insightful considerations, reviewing the thesis and always trying to find the time when I needed guidance. His expert knowledge on Perl has been of great importance when solutions and special Perl functionality have been discussed.

Johan Karlsson who is my instructor at the department of Computing Science at Umeå University has been of great importance, mostly regarding the writing. He has helped me figuring out many important details on how a thesis should be written.

Other persons at TietoEnator contributing with their knowledge, great ideas and reviewing the thesis are the following:
Clas Högvall, who gave me the opportunity to do my thesis work at TietoEnator, has been interested in the work, encouraged me and reviewed the thesis.
Mattias Andersson, one of the Perl gurus beside Gunnar.
The SWDI-team that I have had the opportunity of working with.

I would also like to thank Maria Kroon for reviewing the thesis and the thesis webpage. She has reviewed the language in the thesis and supported me during the work.

GLOSSARY

3rd Party developer Another developer than the original vendor of the development platform.

API See Application Programming Interface.

Application Programming Interface (API) Describes the expected indata, outdata and under what preconditions the interface can be used.

Base class The class from which another class is derived via inheritance.

CORBA Common Object Request Broker Architecture is a set of specifications designed to support platform- and language-independent, object-oriented distributed computing.

Component A composite part with some functionality where the input and output, or characteristics are well-defined.

Dynamic Refers to actions that take place at the moment they are needed rather than in advance.

Extension See Plug-in.

GUI Graphical User Interface, this term refers to a software front-end meant to provide an attractive and easy to use interface between a computer user and application.

Host program An ordinary program[1] with an interface that enables plug-ins to be plugged to it, described in depth in 3.3.

Inheritance The ability to derive a new class from an existing one. Inherited variables and methods of the original (parent) class are available in the new (child) class as if they were declared locally.

IP Internet Protocol - A communication protocol using packet-switching technique to transmit data over the Internet.

Java A set of technologies for creating and safely running software programs in both stand-alone and networked environments.

JavaBeans A portable, platform-independent reusable component model. A component that conforms to this model is called a bean.

MainWindow The window created by Tk that contain other widgets, MainWindow is the first window created in an application.

[1] A definition of an ordinary program is found in section 3.2.2

MIME Multipurpose Internet Mail Extensions, a specification for formatting non-ASCII messages so that they can be sent over the Internet[43]

Module See Component.

OMG Object Management Group, formed in 1989, this consortium of software vendors, developers, and users promotes the use of object-oriented technology in software applications.

Perl One of the most popular scripting language in use today. Used for a wide variety of tasks, including file processing, system administration, web programming, and database connectivity.

Pervasive computing Also called ubiquitous computing, pervasive computing is the result of computer technology advancing at exponential speeds – a trend toward all man-made and some natural products having hardware and software.

Plug-in Refers to a program that has an interface enabling it to be plugged into a host program. A plug-in can provide services to other plug-ins and be dependent of services provided by other plug-ins. A plug-in must be able to be loaded by a host program during runtime. A definition of what a plug-in is can be found in section 3.2.1.

Program "An organized list of instructions that, when executed, causes the computer to behave in a predetermined manner." [44]

RMI Remote Method Invocation.

SOAP Simple Object Access Protocol, a lightweight XML-based messaging protocol used to encode the information in Web service request and response messages before sending them over a network.

Tk A GUI library, generally used with TCL by John Ousterhout, but also available from within C or Perl. Tk is available for X Window System, Microsoft Windows and Macintosh. Tk looks very similar to Motif.

UMTS Universal Mobile Telecommunications System is the world's choice for 3rd Generation wireless service delivery, as defined by the International Telecommunications Union (ITU).

UTRAN UMTS Terrestrial Radio Access Network. A term describing the Radio Network Controllers and Node Base stations of a UMTS network.

XML Extensible Markup Language.

Bibliography

[1] 3GPP. 3gpp specification: 25.401. Webpage, 9 Sept. 2004. http://www.3gpp.org/ftp/Specs/html-info/25401.htm.

[2] Adobe Systems Incorporate. Adobe photoshop: Professional photo editing software. Webpage, 14 Sept. 2004. http://www.adobe.com/products/photoshop/main.html.

[3] And, G. F. Umts network element provisioning tool. In *First Joint IEI/IEE Symposium on Telecommunications Systems Research* (IEI, 22 Clyde Road, Dublin 4, 27 nov. 2001).

[4] Anderson, R. J. *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley & Sons, Inc., 2001.

[5] Bartak, R. Plug-in architecture of constraint hierarchy solvers. The Practical Application Ltd., pp. 359–371.

[6] Bochmann, G. V., and Petrenko, A. Protocol testing: review of methods and relevance for software testing. In *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis* (1994), ACM Press, pp. 109–124.

[7] Bolour, A. Notes on the Eclipse Plug-in Architecture. Tech. rep., Bolour Computing, apr. 2004. http://eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html.

[8] Bronsard, F., Bryan, D., Kozaczynski, W. V., Liongosari, E. S., Ning, J. Q., Olafsson, A., and Wetterstrand, J. W. Toward software plug-and-play. vol. In Proceedings of the 1997 symposium on Software reusability, pp. 19 – 29.

[9] Budd, T. A. *Classic Data Structures In C++*, first ed. Addison Wesley, 1994.

[10] Coulouris, G., Dollimore, J., and Kindberg, T. *Distributed systems: concepts and design*, third ed. Addison-Wesley Longman Publishing Co., Inc., 2001.

[11] Firefox. Firefox help: Firefox faq. Webpage, 31 aug. 2004. http://texturizer.net/firefox/faq.html#q3.1.

[12] France Telecom SA. ASN.1 information site, 7 okt. 2004. http://asn1.elibel.tm.fr/ .

[13] FRANZ, M., FRÖHLICH, P. H., AND KISTLE, T. Towards language support for component-oriented real-time programming. In *Proceedings of the Fifth International Workshop on Object-Oriented Real-Time Dependable Systems* (1999), IEEE Computer Society, p. 125.

[14] GOSLING, J., BRACHA, G., JR., G. L. S., AND JOY, B. *The Java Language Specification*, second ed. Addison-Wesley, 2000.

[15] HANDSCHUH, S. Ontoplugins – a flexible component framework. Tech. rep., Institute AIFB, 2001. `http://www.aifb.uni-karlsruhe.de/WBS/sha/papers/ontoplugin.pdf`.

[16] HANDSCHUH, S., STAAB, S., AND MAEDCHE, A. Cream - creating relational metadata with a component-based, ontology-driven annotation framework, 2001. In First International Conference on Knowledge Capture (K-CAP).

[17] HÖLZLE, U. Integrating independently-developed components in object-oriented languages. In *ECOOP* (1993), pp. 36–56.

[18] HUMPHREY, W. S. *A Discipline for Software Engineering*, first ed. Addison-Wesley Publishing Company, Carnegie Mellon University, United States, 1995.

[19] KICZALES, G., LAMPING, J., LOPES, C. V., MAEDA, C., MENDHEKAR, A., AND MURPHY, G. C. Open implementation design guidelines. In *International Conference on Software Engineering* (1997), pp. 481–490.

[20] KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C. V., LOINGTIER, J.-M., AND IRWIN, J. Aspect-oriented programming. vol. In Proceedings of the Europpean Conference on Object-Oriented Programming(ECOOP), Springer-Verlag LNCS 1241.

[21] KLEINER, A., AND BUCHHEIM, T. A plugin-based architecture for simulation in the f2000 league. vol. In: Proceedings of the International RoboCup Symposium '03.

[22] KUROSE, J. F., AND ROSS, K. *Computer Networking: A Top-Down Approach Featuring the Internet.* Addison-Wesley Longman Publishing Co., Inc., 2002.

[23] LEWIS, J., AND LOFTUS, W. *Java – Software Solutions*, second ed. Addison Wesley, 2000.

[24] LIDIE, S., AND WALSH, N. *Mastering Perl/Tk: graphical user interfaces in Perl.* O'Reilly & Associates, Inc., 2002.

[25] MAGEE, J., AND KRAMER, J. Dynamic structure in software architectures. In *Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering* (1996), ACM Press, pp. 3–14.

[26] MEZINI, M., AND LIEBERHERR, K. Adaptive plug-and-play components for evolutionary software development. In *Object-Oriented Programming Systems, Languages and Applications Conference,* in *Special Issue of SIGPLAN Notices* (Vancouver, October 1998), C. Chambers, Ed., no. 10, ACM, pp. 97–116.

[27] MOZILLA. Tutorial: Creating a mozilla extension. Webpage, 26 aug. 2004. http://www.mozilla.org/docs/tutorials/tinderstatus/.

[28] NETSCAPE COMMUNICATIONS CORPORATION. Plug-in guide. Webpage, 26 aug. 2004. http://developer.netscape.com/docs/manuals/communicator/plugin/index.htm.

[29] NETSCAPE COMMUNICATIONS CORPORATION. Understanding server plug-ins, 15 Sept. 2004. http://developer.netscape.com/docs/manuals/directory/plugin/intro.htm.

[30] OBJECT TECHNOLOGY INTERNATIONAL, I. Eclipse platform – technical overview. Tech. rep., Object Technology International, Inc., feb. 2003. http://www.eclipse.org/whitepapers/eclipse-overview.pdf.

[31] OXYGEN SOFTWARE. Oxygen software, 15 Sept. 2004. http://www.oxygensoftware.com/.

[32] PETINOT, Y., GILES, C. L., BHATNAGAR, V., TEREGOWDAE, P. B., AND HAN, H. Enabling interoperability for autonomous digital libraries: an api to citeseer services. In *Proceedings of the 2004 joint ACM/IEEE conference on Digital libraries* (2004), ACM Press, pp. 372–373.

[33] PFLEEGER, S. L. *Software Engineering, Theory and Practice*, second ed. Prentice-Hall, Inc., Upper Saddle River, NJ 07458, United States, 2001.

[34] PINTO, M., FUENTES, L., FAYAD, M. E., AND TROYA, J. M. Separation of coordination in a dynamic aspect oriented framework. In *Proceedings of the 1st international conference on Aspect-oriented software development* (2002), ACM Press, pp. 134–140.

[35] RAN, A. Software Isn't Built From Lego Blocks. vol. Proceedings of the 1999 symposium on Software reusability, pp. 164–169.

[36] SCHWERN, M. G. search.cpan.org: Michael g schwern / class-virtual-0.04, 7 okt. 2004. http://search.cpan.org/~mschwern/Class-Virtual-0.04/ .

[37] SITEDESIGNER TECHNOLOGIES, I. Api and plug-in architecture - entend 3d-ftp with your own plug-ins. Webpage, 1 Sept. 2004. http://www.3dftp.com/api.htm.

[38] SOAPUSER.COM. Soap basics 1 : What is soap? Webpage, 9 Sept. 2004. http://www.soapuser.com/basics1.html.

[39] STRAND, G. Master's thesis proposal – load module execution/trace & error analyzer tool. Paper, 2 apr. 2004.

[40] TORKAR, R. Dynamic Software Architectures. In *Chapter 3: Architectiong Component-Based Systems* (2002), pp. 21–28.

[41] WALL, L., CHRISTIANSEN, T., AND ORWANT, J. *Programming Perl*, third ed. O'Reilly Media Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 7 2000.

[42] WALLNAU, K., STAFFORD, J., HISSAM, S., AND KLEIN, M. On the relationship of software architecture to software component technology. vol. In Proceedings of the 6th International Workshop on Component-Oriented Programming (WCOP6).

[43] WEBOPEDIA. What is mime? - a word definition from the webopedia computer dictionary. Webpage, 7 Sept. 2004. http://www.webopedia.com/TERM/M/MIME.html.

[44] WEBOPEDIA. What is program? - a word definition from the webopedia computer dictionary. Webpage, 27 aug. 2004. http://www.webopedia.com/TERM/P/program.html.

[45] WEBOPEDIA. What is script? - a word definition from the webopedia computer dictionary. Webpage, 27 aug. 2004. http://www.webopedia.com/TERM/S/script.html.

[46] WECK, W. Independently extensible component frameworks. vol. In Proceedings of the International Workshop on Component-Oriented Programming (WCOP96). In M. Muhlhauser, editor, Special Issues in Object-Oriented Programming - ECOOP96 Workshop Reader. dpunkt Verlag, Heidelberg, 1997.

[47] WEST, C. H. Protocol validation in complex systems. In *Symposium proceedings on Communications architectures & protocols* (1989), ACM Press, pp. 303–312.

Appendix A

GUI - Prototypes

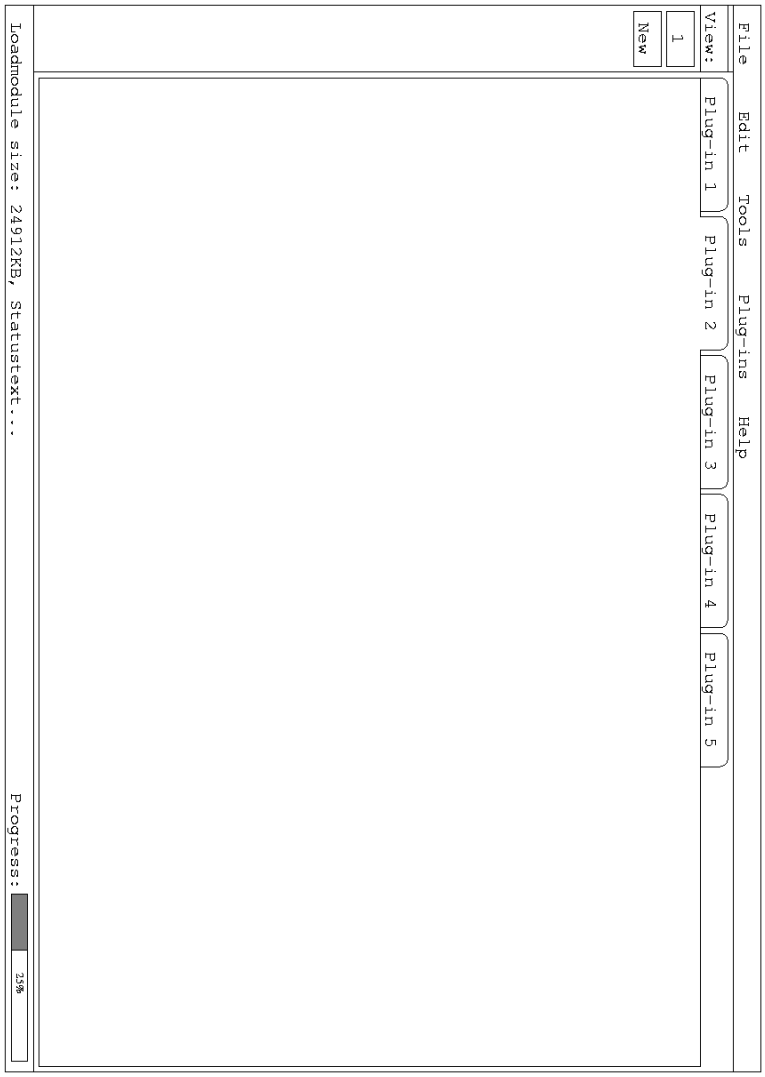

Figure A.1. First prototype of the GUI, created in 2004-05-14.

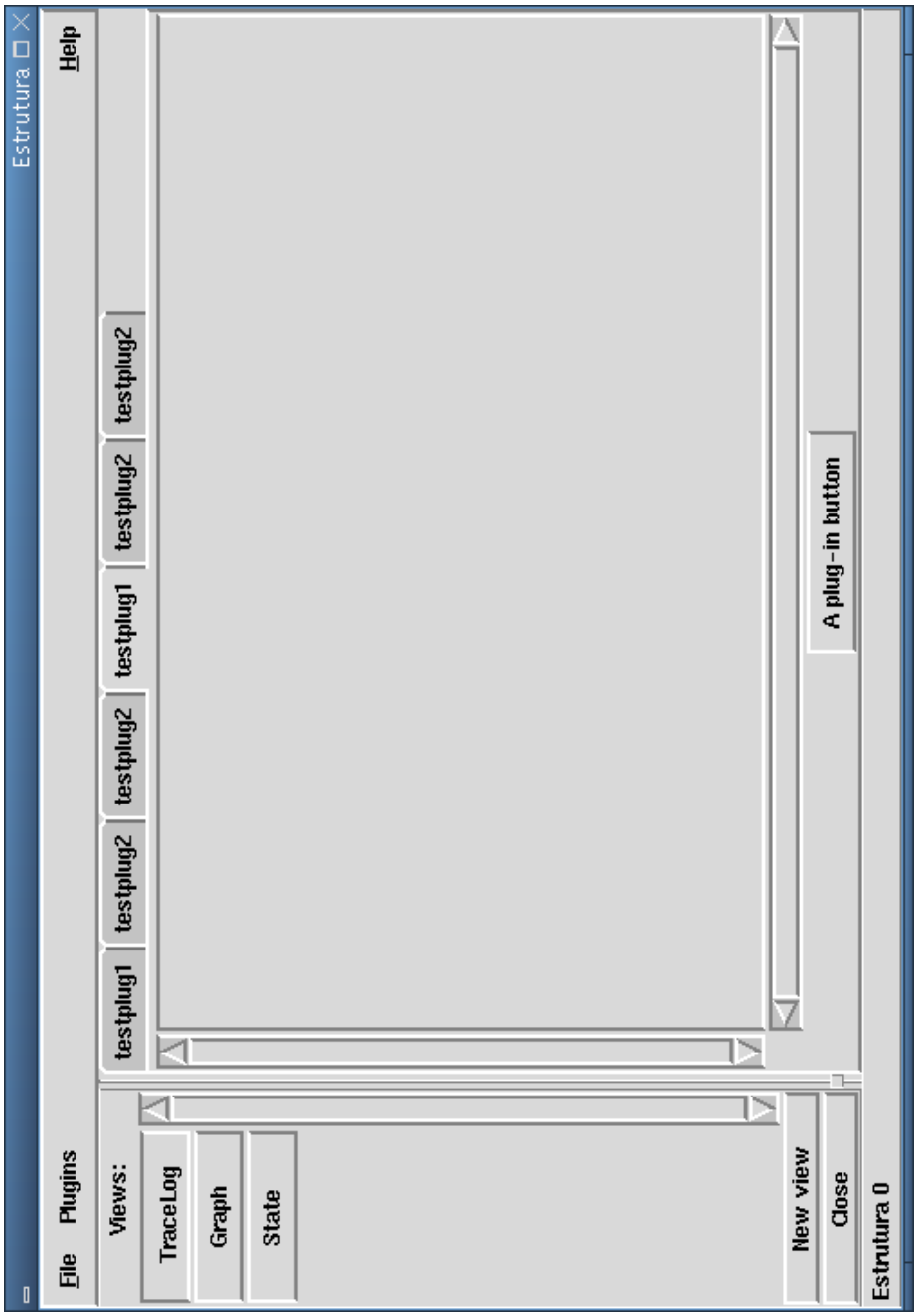

Figure A.2. Second prototype of the GUI, created in 2004-08-11.

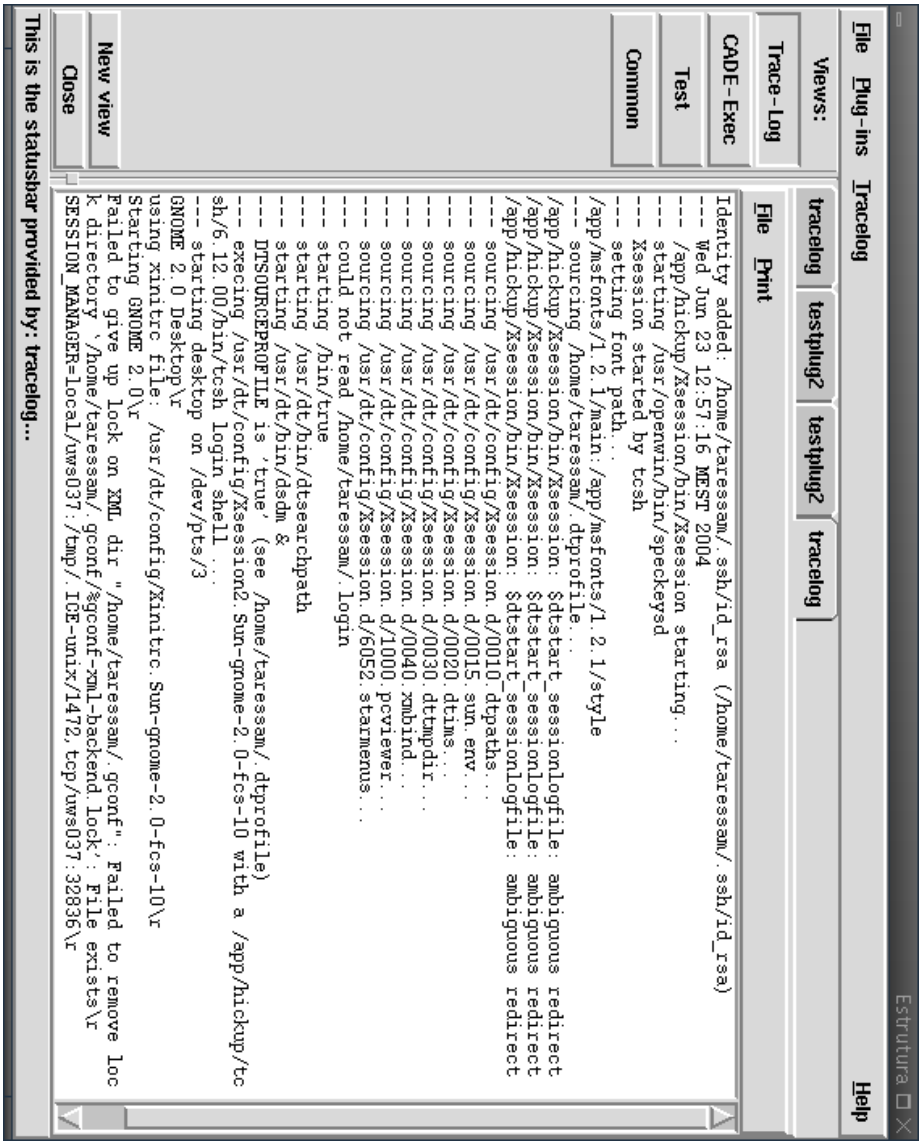

Figure A.3. Screenshot of the final version of Estrutura.