
Computer Science

---

Johan Thorbjörnsson

Peter Svensson

# Component Based Graphical User Interface

---

Bachelor's Project

2000:11

# Component Based Graphical User Interface

Johan Thorbjörnsson

Peter Svensson

© 2000 The authors and Karlstad University

This report is submitted in partial fulfillment of the requirements for the Bachelor's degree in Computer Science. All material in this report which is not our own work has been identified and no material is included for which a degree has previously been conferred.

_____

Johan Thorbjörnsson

_____

Peter Svensson

Approved, 2000-05-31

_____

Advisor: Nils Dåverhög

_____

Examiner: Stefan Lindskog

## Abstract

The background of the thesis is that Ericsson Infotech (EIN) today has a simulation product (SEA) that is built using components. The components are combined at run-time to create a simulation of the system the user needs. The system is divided in a simulation part and a control part. The component system used only covers the simulation parts not the graphical user interface (GUI) used to control the system.

In this thesis we have evaluated some existing technologies that can be used to build a GUI that is run-time extensible using some form of component structure. We propose a technology that are suitable for EINs needs. We have also built simple prototypes using the selected technologies.

The general solution to the problem is divided into two parts, dynamic extension of functionality and comprehensive window control. These two problems are analyzed separately for each technology. EIN has stated the following technologies to analyze: Tcl/Tk, Java, KDE and GNOME. For each technology/language a distributed and a non distributed technology is analyzed.

All the distributed technologies give a overhead and a high level of complexity that is not needed in this application, therefore the non distributed technologies is selected. The selected technologies to implement are:

- Tcl/Tk using the source command and namespaces
- Java using dynamic class loading.
- KDE2 using the KPart technology.

Finally the technology and language that we recommend to use for the development of a Component Based Graphical User Interface is Tcl/Tk using namespaces or Java using dynamic class loading. The selection of these technologies is based on the analysis and the implementation of the different technologies.

# Table Of Contents

# List of Figures

# List of Tables

# 1. Introduction

The background of the thesis is that Ericsson Infotech (EIN) today has a simulation product that is built using components. The components are combined at run-time to create a simulation of the system the user needs. The system is divided in a simulation part and a control part. The component system used only covers the simulation parts not the graphical user interface (GUI) used to control the system. This leads to problems with trying to keep the GUI updated with all the different simulation components designed by EIN and third party providers. To solve this problem EIN would like to have a GUI system that is extensible a run-time so that a component can consist of a simulation part and an optional GUI part that adds functionality to the GUI.

The goal of the thesis work is to propose a suitable technology for designing component based graphical user interfaces. In other words, to propose a technology to build an application that is possible to extend with unknown modules, unknown at build time, after the application is built.

In this thesis we have evaluated some existing technologies that can be used to build a GUI that is run-time extensible using some form of component structure. Describe the pros and cons of the different solutions. Finally we propose some technologies that are suitable for EINs needs. We have also built simple prototypes using the selected technologies.

The second chapter describes the background of the problem to this thesis. The Simulator Environment Architecture (SEA) is described, what it is and a overview description of how it works. At the end of this chapter we state the problems to solve and the limitations of the thesis.

In the third chapter a general analyze of the problems is done. There are several ways to solve the control part problem of SEA, problems which has to do with the supervision and graphical presentation of the different modules in the SEA. The problems may be divided into two separate parts: dynamic extension of functionality and comprehensive window control. The solution for these problems are discussed in this chapter.

In chapter four we state the different technologies to be evaluated. The different technologies is analyzed considering how the technology in question can be used to solve dynamic extension

of functionality and comprehensive window control. At the end of each section for each technology some general aspects of that technology is considered. Finally the technologies to implement are selected.

Chapter five is the description of the implementation of the selected technologies. First the Tcl/Tk implementation is described using the Tcl source command, second Java using dynamic class loading and finally KDE2 using KParts is described.

Chapter six is the conclusion of the thesis. In this section we state the technology that we find the most suitable to solve the problem of a component based graphical user interface.

Chapter seven is the reference list. This chapter states the references used in the thesis.

# 2. Background

This section describes the background of this thesis. First a description of Ericsson as a company, Ericsson Infotech (EIN) and the department Test And Simulated Platform (TSP), as they describe them selves. Then the Simulator Environment Architecture (SEA) is described, what it is and a overview description of how it works. Finally we state the problems to solve and the limitations of the thesis.

## 2.1. Telefonaktiebolaget LM Ericsson

Ericsson is a world-leading supplier in the fast-growing and dynamic telecommunications and data communications industry, offering advanced communications solutions for mobile and fixed networks, as well as consumer products. Ericsson is a total solutions supplier for all customer segments: network operators and service providers, enterprises and consumers. Ericsson has more than 100,000 employees, representation in 140 countries and clearly the world's largest customer base in the telecommunications field. [1]

## 2.2. Ericsson Infotech (EIN)

Ericsson Infotech AB, located in Karlstad, with over 550 employees is a product and development company in the field of mobile telecommunications. EIN has product and development responsibility within a number of product area, including Signalling System No.7 (SS7) and protocol converters, APZ emulators and simulators, wireless Internet solutions, radio net products, as well as maintenance and customer support systems. [2]

## 2.3. Department of Test Support and Simulated Platforms (TSP)

The departments goal is to become Ericsson's leading supplier of simulator products as regards platforms, systems, and network solutions. The mission is to offer products and services, based on APZ Emulators and Simulators, for Ericsson's customers to improve business and within Ericsson to reduce costs. [3]

## 2.4. Simulator Environment Architecture

Today Ericsson Infotech (EIN) has a simulation product called Simulator Environment Architecture (SEA) that is used to simulate different complex system, in this case an AXE switch. SEA is built using different components that each simulates different parts of the system. The components are combined at run-time to create a simulation of the system the user needs. The system is divided in a simulation part and a control part as shown in figure 1. The separate parts are discussed in detail below. The component system now used only covers the simulation parts and not the graphical user interface (GUI), the control part, used to control the system.

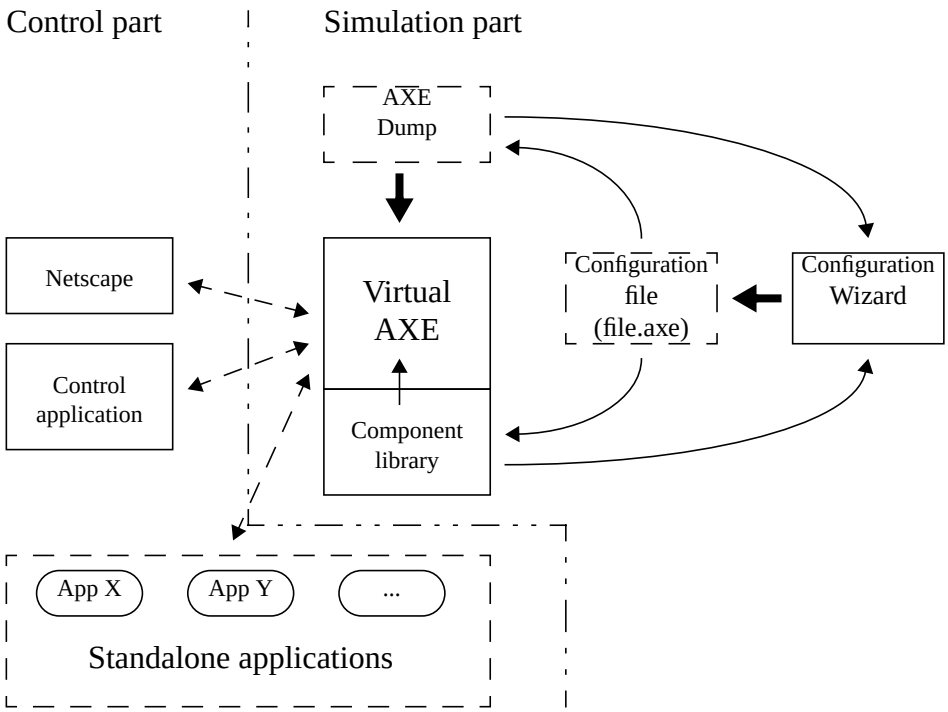


Figure 1: SEA Overview

### 2.4.1. Simulation Part

The simulation part of SEA consists of a number of interacting components. The configuration file contains information about everything needed to build the AXE, such as software (dump) and hardware information. In the configuration file, the name and location of the dump is indicated, and also how the AXE will behave once the dump is loaded. The needed hardware such as different “physical” components and how they are connected are also indicated. The SEA Configuration Wizard is used to create a configuration file. The dump is loaded into the Wizard and as the Wizard reads the dump, it consults the component library to check for available

“hardware” components. The component library as a whole is a part of SEA and contains every component so far developed. The configuration file points out the dump and components to be loaded. When the dump and the components are loaded the virtual AXE is complete. The SEA Control Center, Netscape and other applications can now be used to work with our virtual AXE.

### 2.4.2. Control Part

Today the different components in SEA are controlled in different ways, for example there is a application called AT-console which is a text based tool used to communicate with the virtual AXE and its currently loaded components, hereafter named SEA core. While the AT-console can be used to communicate with several different types of components in the SEA core there are also some components that have their own component specific control applications, an example of this is the graphical component that simulates a telephone which is connected to a LIC component in the virtual AXE.

The main application for monitoring and controlling SEA is the SEA Control Center, shown in figure 2. The SEA Control Center is used to control and monitor the virtual AXE itself. From the SEA control center the different component control applications like the AT-console and other graphical control applications can be started.

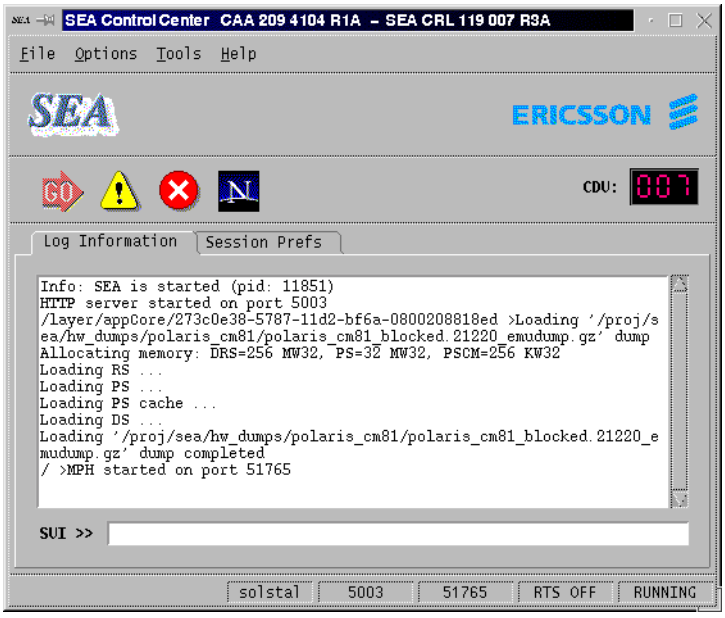


Figure 2: SEA Control Center

All external graphical user interface (GUI) components, like the SEA Control Center and the LIC-module, communicates with the SEA core using messages between GUI components and named SEA core entities (instance of a component). To handle the framing and routing of this messages there is a protocol named MPH (Message Protocol Handler). The MPH offers a multiplexed socket, allowing 255 different channels (virtual TCP/IP sockets) on one socket, each channel is used to communicate with a named SEA entity. MPH libraries are implemented in C, Java and Tcl/Tk by EIN/TSP. Figure 3 shows the MPH connection in between the virtual AXE and the different components which it communicate with.

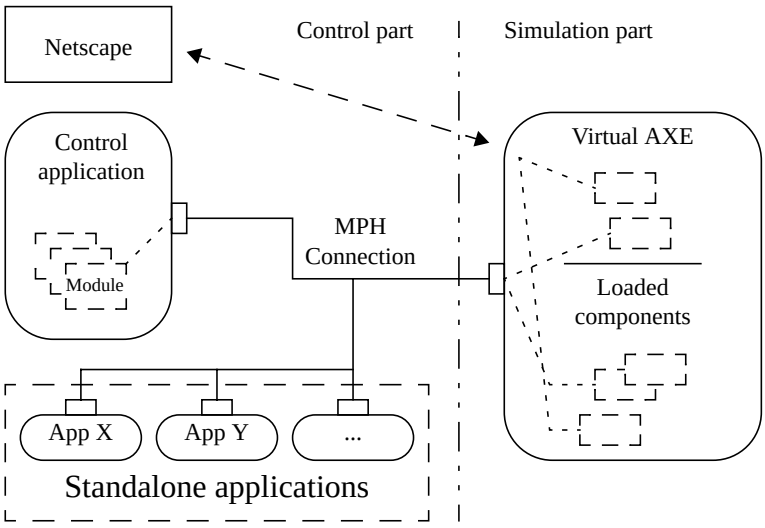


Figure 3: SEA MPH Connection

Even if the AT-console and the graphical modules for each component that is loaded into the SEA core can be started from the SEA Control Center they still are stand alone components or a part of the SEA Control Center that runs in a separate toplevel window.

The SEA core contains a web server and most of the components in the virtual AXE support functionality to communicate with a web browser for viewing and controlling data in the component. This method of controlling the SEA core components works, but is not flexible enough. TSP also wants the control and viewing of the components in a standalone application.

## 2.5. Control Application Problems

New components to SEA are under constant development, both by EIN and by third party developers. One problem with the SEA Control Center, as it works today, is that when a new component is developed there is no way to extend the existing SEA Control Center so it can be

used to control or start the new GUI component without altering the source code for it. In other words, there is no way to just add a new GUI component to the SEA Control Center in the same way as it is possible to add a new component to the component library in SEA. This leads to problems while trying to keep the GUI updated with the new components that is being developed. The SEA Control Center either have to be remade for each new component or a new component must provide it's own stand alone GUI.

SEA gets the information about which components to load for a specific configuration from a configuration file at startup, but it is also possible to add new components to a SEA configuration at runtime. The existing SEA Control Center does not support this kind of dynamic loading of GUI applications.

Since the control applications for the different components all run in their own toplevel windows there is a problem if there are many components that needs to be supervised. The screen gets full with windows and it gets difficult to keep track of them all. Figure 4 shows the SEA Control Center and just a few graphical modules needed to control the simulation.

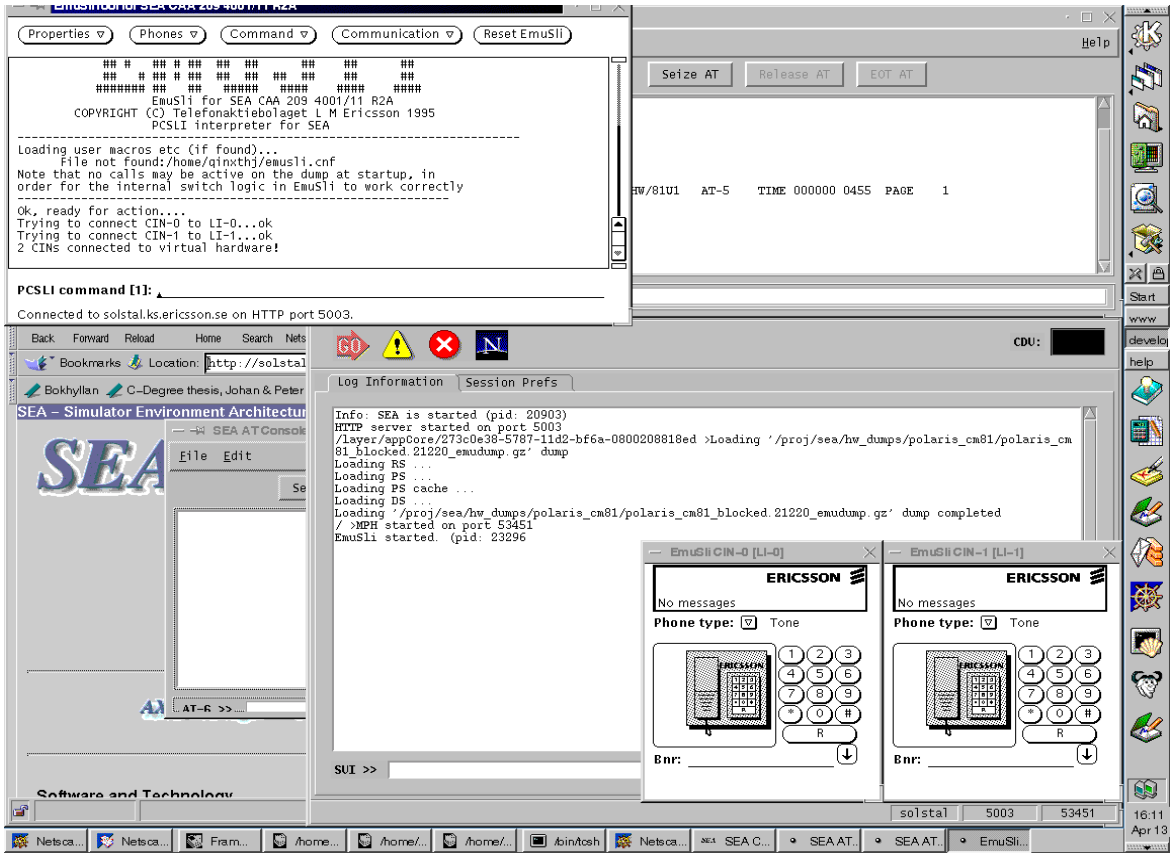


Figure 4: SEA Control Center and a few other needed applications.

In conclusion, the main problems with the control part of SEA is that the GUI must be remade when a new component is developed, the GUI can not be extended at runtime and controlling many components generates lots of toplevel windows.

## 2.6. Aim of Thesis

The aim of this thesis is to evaluate the existing technologies that can be used to build a GUI that is run-time extensible using some form of component structure. Describe the pros and cons of the different solutions. Propose some technologies that are suitable for EIN's needs and to build a simple prototype for each selected technology.[1]

In other words, the purpose of the thesis is to look into the problems with the control part of SEA that has to do with the supervision and graphical presentation of the different modules in the SEA core. The focus of this thesis will be separated in to two parts. First, how is **dynamic extension of functionality** best solved using different technologies to support new modules without recompiling the complete control part of SEA. Second, look at different technologies that support some kind of technical solution for the problem with to many top level windows, that is to get a **comprehensive window control**.

## 2.7. Limitations

The SEA simulation and control applications runs on a Sun Solaris Unix platform today, and will continue to do so. There has been some work porting parts of the SEA to a Linux/PC platform, but this is not currently working. There is also a possibility to port and run control applications on any platform, i.e Linux/PC, Windows NT/PC etc., but the SEA core still needs to run on a Sun Solaris platform. The different parts then may communicate through the MPH.

The graphical user interface as in the look and feel is not considered. That is, the usability of the application is not considered, only the technology to provide a dynamic base to build upon.

1. The official description of the thesis is enclosed in appendix B.

## 2.8. Summary

Ericsson Infotech (EIN) has a simulation product called Simulator Environment Architecture (SEA) that is used to simulate different complex system, in this case an AXE. SEA is divided in a simulation part and a control part.

The main problems with the control part of SEA is that it must be remade when a new component is developed, the GUI can not be extended at runtime and controlling many components generates lots of toplevel windows.

The aim of this thesis is to evaluate the existing technologies that can be used to build a GUI that is run-time extensible using some form of component structure. Describe the pros and cons of the different solutions. Propose some technologies that are suitable for EINs needs and to build simple prototypes using the selected technologies.

# 3. General Solution

This section describes the general solution to solve the stated problems in the previous chapter. There are several ways to solve the control part problem of SEA, problems which has to do with the supervision and graphical presentation of the different modules in the SEA. As stated in chapter 2.6, Aim of Thesis, the solution to solve the problem will be divided into two parts, dynamic extension of functionality and comprehensive window control. The solution for these problems are discussed in this chapter.

## 3.1. Dynamic Extension of Functionality

The problem of dynamically extending the Main Control Applications functionality can be solved through a modular design of the Main Control Application so that it is possible to add functionality to it without changing the already existing code. By using modular design with a standard interface the Main Control Application could load new or changed modules to extend its functionality. In this way a new component can be supported by the Main Control Application just by adding a new module to it.

In practice this modular design could be done by implementing the control functionality for each component in separate modules and then have the Main Control Application ‘ask’ the SEA core about which components that are loaded in the current SEA configuration. Then the Control Application checks if the different types of components have a associated functionality module. If so, the Main Control Application then loads the associated modules and in this way extend its functionality. Each module must provide the specific functionality that is needed to manage its associated component. Figure 5 shows the dynamic loading of modules and the interaction between a given components and its module.

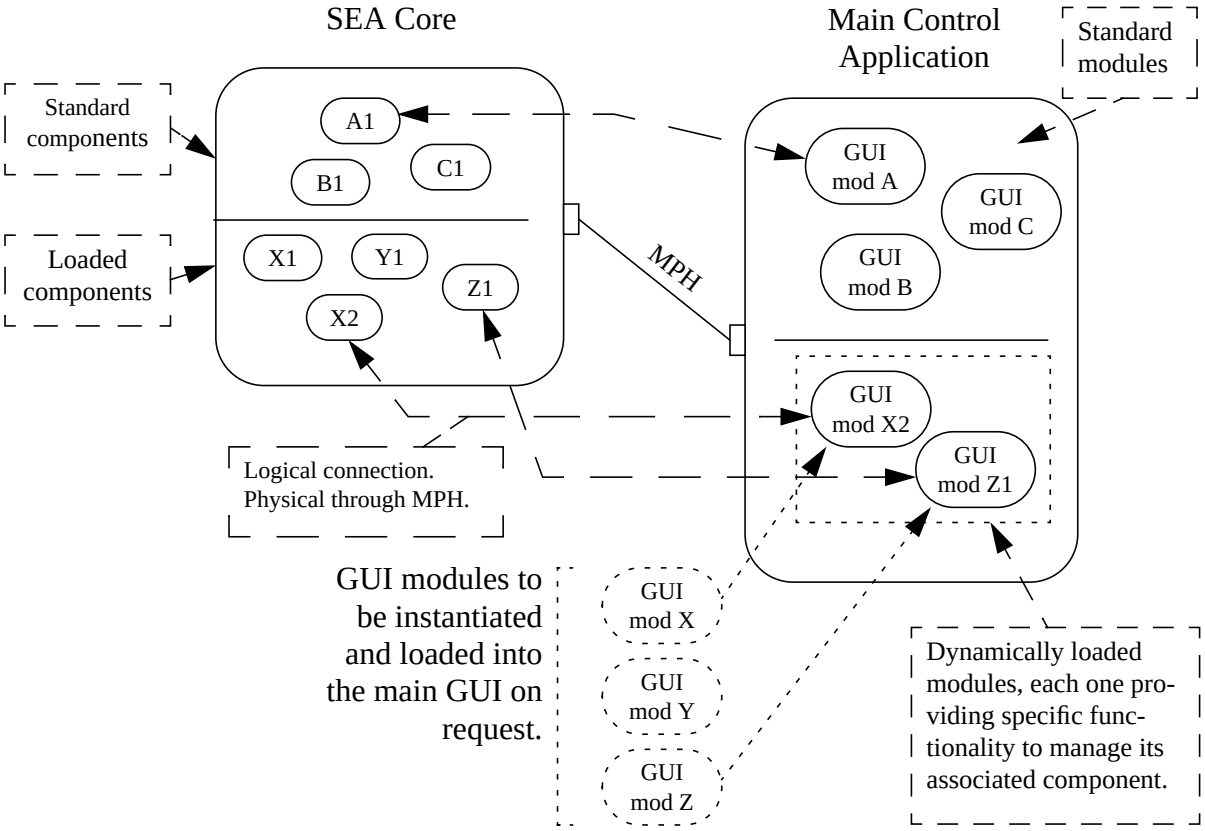


Figure 5: Dynamic extension of functionality

For each module that is instantiated and loaded into the SEA core one must know if the loaded module does or does not have a graphical module associated to it, the name of the module and where to find it. Once this is known the module can be loaded in runtime into the Control Application when needed. When the module is not wanted anymore it can be deallocated to save system resources.

Once a new module is loaded there must be a way for the Main Control Application to initiate and to communicate with the loaded module in a known way. In practice this means that each module must support some predefined functionality. The module has to implement a set of standard methods like initiation of the module, drawing the module, callback functions for communication, etc. The names of the standard methods also have to be defined, for example compare to a Java Interface.

By using modular design the Control Application can be implemented so that it is possible to add new functionality to the Control Application not only at startup but also at runtime if the SEA configuration is changed while running.

In conclusion the dynamic extension of functionality must include the following steps: first, the components that are currently loaded in the SEA core has to be identified. The second step is to find out if the loaded components does have a GUI module or not. Finally, the functionality of the matching modules have to be added to the main application. These steps are described in the following subsections.

### 3.1.1. What Components are Currently Loaded

After the SEA-core has started, components are loaded and configured according to the configuration file, it is time for the Control Application to start. The first thing the Control Application needs to find out is which the loaded components in the SEA core are. The SEA core has to support functionality, through MPH or some other connection, that gives information about all the loaded component types and the instances of it to the Control Application.

### 3.1.2. Does the Loaded Component Type Have a GUI Module

Once knowing the instances of the different components the Control Application has to find out if a given component has a GUI module or not, the name of it and were it is located. If there exists a corresponding module to a given component it has to be loaded. There are a few technical approaches to solve this problem. Two general solutions will be considered, using some sort of distributed technology, like CORBA or Java RMI, and a simpler solution not involving distribution, like Tcl source command or KDE Kpart.

#### 3.1.2.1. Distributed Solution

Using a distributed solution gives the advantage of having instances of modules anywhere on a network, that is the Main Control Application does not need to be loaded full of different modules and their code, it will just make a remote call to access the functionality. Distribution also gives the possibility to have a central for all modules, thus giving the advantage of easily updating and maintaining the modules.

In principal the distribution of modules can be implemented as shown in figure 6.

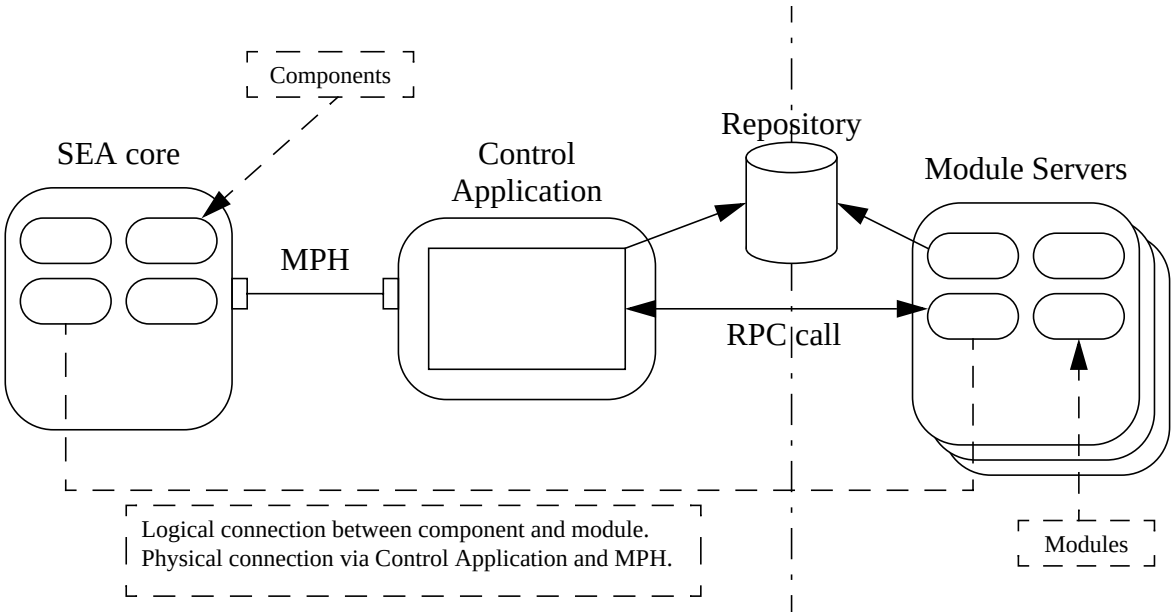


Figure 6: Distributed solution

The module servers contains the instances of the available modules. The repository is a database containing information about all available modules. In order for the Control Application to locate a specific module, the module has to be registered in the repository. That is, when a new module is created it has to register its name and location. When a specific module is wanted the Control Application asks the repository for that specific modules location. When the location of a specific module is known it is possible for the Control Application to incorporate the functionality of the remote module.

#### 3.1.2.2. The Non Distributed Solution

In this case the functionality of the repository has to be handled by either the SEA core or the Control Application. Some different approaches can be taken. In either case the loaded components in the SEA core are known.

- Increasing the functionality of each SEA core component and the MPH is one solution, where each component will have a method for asking whether it has or has not a graphical module, if so the name of it and where to look for it.
- The main GUI has to check each component for its type, check for the found component type in a separate GUI configuration file, if the component type has a GUI module, the name of it and where it is located.

There are of course a number of solutions but these are the most reasonable, the latter is probably the easiest to implement for test, because there is no changes needed for the SEA core.

## 3.2. Comprehensive Window Control

The second part considers the problem of having too many top level windows at the same time. This is solved by making the Main Control Application offer a “drawing area” for each graphical module that is loaded to draw its graphics within. One way is to have a Main Control Application, where each separate module has its own child window, within the main window. Each module is then free to use the given child window for its graphical user interface, like a multi document word processor application. Another way, not using child windows, is using tabbed panes. The Main Control Application holds the tabbed pane, then each pane is the drawing area offered to each module to draw its graphical user interface within. The tabbed pane solution will most likely give a more controllable GUI when having many GUI modules open at the same time. Yet another solution is to use some kind of tree view, one node for the module type and the instances of the module as subnodes. When an instance is selected the instance is given a drawing area to draw its graphical control within. Regardless if the GUI module is in a child window, a pane or a tree view, the module should be able to be torn off from the main application, to an own separate top level window.

### 3.2.1. Selecting and Showing Instances

When all the instances of the different loaded modules are known, and one is to be selected they must be graphically represented in some way. As mentioned in chapter 2.7 the graphical issues as in look and feel is not considered, but there are some basic graphical alternatives namely:

- The Main Control Applications menubar is extended with a item “module” containing all the module types loaded. Each module type then has a submenu attached to it with all its instances, like menubar -> module type -> instance. This will cause some troubles when there are to many module types or instances of one particular type, since the list of all items will fill up the screen.
- Another approach is using a File dialog type. A child window like “open file” is created having the module type as the directory tree and the instance like the files.
- Yet another approach is having the modules and instances in a tree view, one node for the module type and the instances of the module as subnodes.

### 3.2.2. Drawing Area for the Selected Module

Once a module is selected a new drawing area must be created by the Control Application. This area is given, pointed out, to the module upon its creation. The module then uses this drawing area to draw its graphical components within. The given drawing area can be contained in a few different ways by the Control Application, for example a tab in a tabbed pane or a child window in a container window, figure 7.

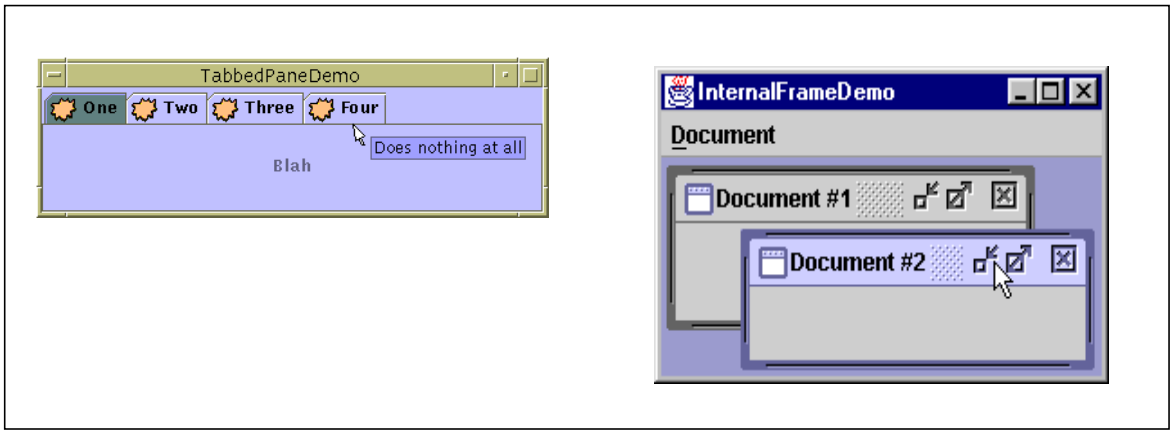


Figure 7: Tabbed pane and container window.

Figure 7 shows the Java Swing graphical classes. This is just to illustrate some different solutions for the drawing area that can be provided for the modules.

### 3.2.3. Extending the Menubar

Finally when a selected instance of a module is graphically drawn in its given area, the Control Applications menubar should support methods to extend the functionality of the menubar for the active module. In the same way as the Control Application gives a drawing area to the module, the Control Application will give, point out, where to extend the menubar.

## 3.3. Summary

The general solution is divided into two parts, dynamic extension of functionality and comprehensive window control. The dynamic extension of functionality can in turn be divided into two subparts, a distributed solution and one that is not distributed.

Dynamic extension of functionality

To extend the functionality of the main GUI application a few steps has to be considered.

- *What components are currently loaded*

  Find out the components types and the instances currently loaded in the SEA core.
- *Does the loaded components have a GUI module*

  One has to know if a component in the SEA core has graphical module or not. Each module has to be extended with this functionality or a separate GUI-configuration file can be used.

Comprehensive Window Control

Letting the main GUI application contain the different modules graphics. There are a few steps needed to achieve this.

- *Selecting and showing instances*

  To show a particular instance it has to be selected from some list containing the modules and the currently loaded instances of it. This is graphically represented in three basic ways, menubar, File dialog child window or a tree view.
- *Drawing area for the selected module*

  Once an instance of a particular module type is selected, the main GUI application has to offer a drawing area that is given, pointed out to the module.
- *Extending the menubar*

  Finally, the main GUI applications menubar can be extended with functionality from the loaded module. In the same way as the Control Application gives a drawing area to the module, the Control Application will give, point out, where to extend the menubar.

# 4. Analysis of some Chosen Technologies

In this section some different technologies are analyzed according to the general solution. First the technologies to be considered are stated. The following subchapters analyze the different technologies considering how the technology in question can be used to solve dynamic extension of functionality and comprehensive window control. At the end of each section for each technology some general aspects of that technology are considered. Finally the technologies to implement are selected.

## 4.1. The Technologies to be Considered

Comparison of some chosen technologies to solve both the problem of dynamic extension of functionality and the problem of comprehensive window control. The following are the technologies to be considered:

Table 1: Technologies to be considered

| Technology | Provided solution |
|---|---|
| Tcl/Tk | Tcl Source command, Tcl-DP |
| Java | Dynamic class loading, RMI |
| KDE | Mico Orb, KParts |
| Gnome | ORBit, Bonobon |

Tcl/Tk is a natural choice since most of the GUI applications for SEA is written in Tcl/Tk today, and EIN/TSP has a wide experience in Tcl/Tk development. Java is a object oriented, flexible language suited for graphical development with multi platform support through its binary code and virtual machine solution, all this could be a great advantage for development now and in the future. KDE and Gnome are selected because they stand for the new generation of desktop environment in the UNIX community.

## 4.2. Tcl/Tk

Tcl/Tk is divided into two parts. The first part is Tcl, pronounced tickle, stands for Tool Command Language and was created by John Ousterhout in 1987. Tcl is an interpreted language and is more like a scripting language than a programming language, so it shares greater similarity to the C shell or Perl than it does to C++ or C. Tcl provides generic programming facilities, such as variables and loops and procedures. Since Tcl is an interpreted language, it of course executes slower than compiled C code, but still it is surprisingly fast. [4]

The other part of Tcl/Tk is Tk. Tk is a graphical user interface toolkit that makes it possible to create powerful GUIs quickly. Tk extends the built-in Tcl commands with commands for creating and controlling graphical user interface elements called widgets. A widget can be a button, text window, scrollbar etc.

### 4.2.1. Source Command

The dynamic extension of functionality problem can be solved using Tcl's source command. Since Tcl is an interpreted language it is possible to extend the loaded code while executing. The source command takes the contents of a specified file and extends the existing code. This makes it possible to load new modules in a way that solves the problem dynamic extension of functionality.

To get the Main Control Application to interact with the loaded modules, the modules have to support a set of standard methods, procedures in Tcl. Since each module has its own implementation of the given standard procedures, there will be a problem when more than one module is loaded. In that case there will be more than one procedure using the same procedure name. Tcl is not a object oriented language and therefore this problem can not be solved using any class abstraction[1]. A solution to this problem is using Tcl's namespace. A namespace is a collection of commands and variables. It encapsulates the commands and variables to ensure that they will not interfere with the commands and variables of other namespaces. Figure 8 shows an example code of Tcl source and namespace commands.

1. Even though Tcl/Tk is not object oriented there exists packages that extend Tcl/Tk to support object orientation. Analyzing these packages is beyond the scoop of this thesis.

```
# This is a comment in Tcl
# A procedure somewhere
proc loadNewModule {} {
   namespace “moduleName” {
     # This is the new namespace
     source “moduleName.tcl”
   }
}

# Call the procedure to create a new namespace
loadNewModule

# Call a procedure in the new namespace
moduleName::wantedProcedure
```

Figure 8: Tcl source and namespace commands

The procedure `loadNewModule` first creates a new namespace named “moduleName”. In this new namespace, encapsulated by brackets, the new Tcl code is loaded from the sourcefile “moduleName.tcl”. When the `loadNewModule` procedure has been called, it is possible to call the procedures in the new namespace using the :: notation to specify the name of the namespace and one of its procedures.

Tk arranges the widgets in a hierarchical tree structure. Each widget is identified through its location in the tree, this gives the path to the widget from the root. In Tcl the comprehensive window control can be solved by letting the main application create a Tk frame. A Tk frame can be compared to a Java panel. The frames path is passed as an parameter to the module. The given path is then used by the module as the root of its own widget subtree. That is the module draws its widgets from the given path.

### 4.2.2. Extension Packages.

There exists a large number of packages that extends the functionality of Tcl and Tk. A package that extends Tcl and provides a technical solution for the dynamic extension of functionality is Tcl-DP, Tcl Distributed Programming. Tcl-DP is a collection of Tcl commands that simplifies the development of distributed programs. Tcl-DP’s most important feature is a *remote procedure call facility*, which allows Tcl applications to communicate by exchanging

Tcl scripts. For example, the following script shown in figure 9 uses Tcl-DP 4.0 to implement a trivial “id server“, witch returns unique identifiers in response to `GetId` requests.

*Server*

```
package require dp 4.0

dp_MakeRPCServer 1944 tcp0
set i 4
proc getID {} {
   global i
   incr i
}
```

*Client*

```
package require dp 4.0

dp_MakeRPCClient host.domain 1944 tcp0
set id [dp_RPC tcp0 getID]
```

Figure 9: Id server and client using Tcl-DP

The first command executed on both client and server is the Tcl `package` command, which makes Tcl-DP library functions and commands available in the current Tcl interpreter. The server executes the `dp_MakeRPCServer` command, which creates a socket that is waiting for a client to connect. Finally, the server defines the `getID` command, which generates and returns a unique identifier.

The client connects to the server using the `dp_MakeRPCClient` command, which returns a handle that can be used to communicate to the server. Finally, the client invokes the getID command on the server using the `dp_RPC` command. This causes a message containing the command to be evaluated to be sent to the server, where it is evaluated and the results returned.

The Tcl-DP could be used to solve the problem with dynamic extension of functionality. A new process acting as a module server has to be created, shown in figure 6. This server holds all the modules that are dynamically loaded. The advantage of this solution is that the Main Control Application does not need to source the different modules, and therefore need not to concern about any namespace problems. The Main Control Application calls functionality of the different modules using the dp_RPC command. The major disadvantage of this solution is

that it does not really solve the problem, it is just another way to communicate with the different modules. The problem of dynamic loading will now appear in the server instead.

### 4.2.3. General Aspects

*Technical aspects on Tcl/Tk:*

- Tcl/Tk runs on a number of platforms, Windows95/98/NT, Mac and nearly every Unix platform like Solaris, Linux etc.
- Tcl is developed in C and the Tcl interpreter is a library of C functions that implements the Tcl commands and the grammar for the Tcl language. This fact makes it easy to extend the Tcl language with new commands by creating new C libraries. Each command is implemented by one single function in the C libraries. Due to this fact a large number of extension packages to add new sets of Tcl/Tk commands exist. They provide a variety of functionality, like databases, network management and platform specific APIs.
- There is a compiler available to Tcl that translates the Tcl scripts into a bytecode file making it possible to distribute applications without providing access to the original Tcl source code.

*Distribution and licence agreement:*

For distribution Tcl/Tk use open source licence agreement. In short, the open source licence agreement for Tcl/Tk gives permission to use, copy, modify, distribute, and license software and its documentation for any purpose, provided that existing copyright notices are retained in all copies and that this notice is included verbatim in any distributions. No written agreement, license, or royalty fee is required for any of the authorized uses. [5]

## 4.3. Java

Since Java is a widely known language, the language itself will not be described in detail (a complete description of the Java language can be found on http://java.sun.com). However Java has a few aspects that reflects on this work which is wort mentioning. First, Java works with bytecode, which in turn is executed on a virtual machine. This making it possible to use the same bytecode on different platforms, platform independent. This in turn is useful in distributed solutions, where the bytecode can be transferred between or executed on different platforms on the server and client machines. Another aspect, Java is strictly object oriented which makes Java well suited for modular design. In Java it is also easy to create GUI.

### 4.3.1. Dynamic Class Loading

To solve the problem of dynamic extension of functionality in Java it is possible to use the dynamic class loading that is offered by the static method `Class.forName()` in the class `java.lang.Class`. The method `Class.forName()` takes a fully qualified name of a class as a parameter and loads that class into the interpreter and returns a `Class` object for it. In Java there is a `Class` object representing every class loaded into the interpreter, one `Class` object for each class. The `newInstance()` method in class `Class` creates an instance of the class that it represents and returns the newly created instance. Figure 10 shows an example of a dynamically loaded class.

```
// Loads and instanciates class MyClass
java.lang.Class t;
MyClass myClass;

try {
  t = Class.forName("MyClass");
  myClass = (MyClass)t.newInstance();
}
```

Figure 10: Dynamically loaded class in Java.

First a new `Class` object representing `MyClass` is created by the call to `Class.forName`. The `Class` objects method `newInstance()` is then called to create the actual instance of `MyClass`.

The dynamic extension of functionality can be solved using Java's dynamic class loading functionality. Each GUI module has to be implemented as a separate Java class. Since each module has to contain a set of standard methods, the GUI module class has to implement a given standard Java interface. The main application then loads and instantiates those classes that are needed for the current configuration.

In Java, the comprehensive window control can easily be solved by having the main application create and control the panels that the module classes use. That is, when the main application have instantiated a module class, it creates a panel and calls some standard method in the newly instantiated object with the created panel as an argument. The object then uses the given

panel as a base to draw its widgets on. From the objects point of view this panel is its toplevel window.

### 4.3.2. Java RMI

Java supports a few different distributed object technologies, namely Java RMI and CORBA. The Java Remote Method Invocation (RMI) system allows an object running in one Java Virtual Machine (VM) to invoke methods on an object running in another Java VM. RMI provides remote communication between programs written in Java. CORBA on the other hand gives a possibility for Java applications to communicate with objects written in any language that supports CORBA. CORBA solutions will be discussed later on, but not for Java.
Java RMI contains two main parts, clients and a server, figure 11.

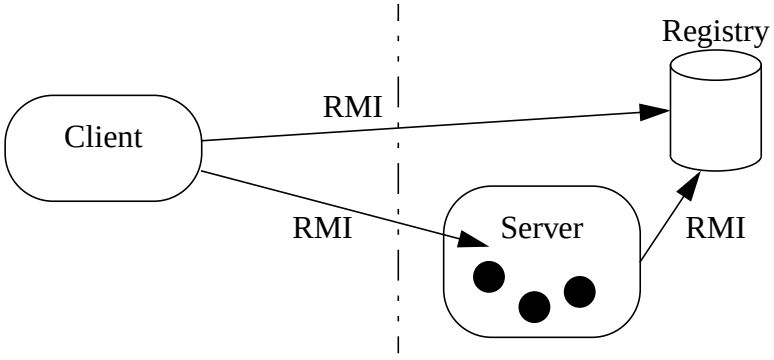


Figure 11: Java RMI

The server application creates and supplies the remote objects. The clients invoke methods on the remote objects by a remote reference. The remote reference is obtained from the registry. RMI provides the mechanism by which the server and the client communicate and pass information back and forth.

One of the central features of RMI is its ability to download the bytecode of an object's class if the class is not defined in the receiver's virtual machine. The type and the behavior of an object, previously available only in a single virtual machine, can be transmitted to another virtual machine, thus extending the behavior of an application dynamically. [6]

RMI could be used to solve the dynamic extension of functionality problem by implementing a distributed solution, figure 6. RMI supports functionality to handle the repository, the registry in Java. That is, functionality to register a new module, get the name and location of a module

etc. Once a reference to a remote module, object, is obtained it is referenced as it was a local object. This high level of abstraction makes it fairly simple to implement a distributed solution in Java.

### 4.3.3. General Aspects

First a short description of the main components and the abbreviations commonly used in the Java community, to avoid confusion.

The Java programming language is currently shipping from Sun Microsystems, Inc. as the Java Development Kit (JDK). All Sun releases of the JDK software are available from the JDK software home page (http://java.sun.com/products/jdk/). Each release of the Java Development Kit (JDK) contains:

- Java Compiler
- Java Virtual Machine
- Java Class Libraries
- Java AppletViewer
- Java Debugger and other tools
- Documentation (in a separate download bundle)

The Java Foundation Classes (JFC) are a comprehensive set of GUI components and services which dramatically simplify the development and deployment of commercial-quality desktop and Internet/Intranet applications. Swing is the project code name for the lightweight GUI components in JFC.

*Technical aspects:*

The JDK 1.1.x Final and 1.2.1 Final is available on these platforms:

- SPARCTM SolarisTM 2.4-2.6
- Intel x86 Solaris 2.5-2.6
- Microsoft Windows 95 / NT 4

The JDK 1.0.2 is available on these platforms:

- SPARC Solaris 2.3-2.5
- Intel x86 Solaris 2.5
- Windows 95 / NT
- Macintosh 7.5

*Distribution and licence agreement:*

The Java Development Kit (JDK) is free to download and use for commercial programming, but not to re-distribute. That is, a source code license is not needed to write and distribute applets or applications in the Java language. Sun's binary license permits developers to write software in the Java language, as well as distribution of the binaries for the Java interpreter along with applications, at no cost. [7]

## 4.4. KDE

KDE is a graphical desktop environment for Unix workstations. It combines ease of use, contemporary functionality and graphical design with the technological superiority of the Unix operating system. [8]

KDE is developed in C++ and the KDE library offers a complete range of widgets, based on the QT widget library, and desktop functionality. The new version of KDE, v2.0, supports a number of interesting new technologies. To name a few, KDE2 offers a technology named KOM/OpenParts, which is a technology built upon the open industry standards such as the object request broker CORBA 2.0. (a complete descripton of the CORBA technology can be found on http://www.omg.org). Another of KDE2s new technologies is KPart. KPart is used to embed applications within existing ones. Both these technologies can be used to solve the dynamic extension of functionality and comprehensive window control in SEA.

Note, KDE2 is only available as a pre-alpha release. KDE 2.0 is scheduled to be released in the spring/summer of 2000. This fact makes the analyzing a bit complicated since it is not given whether the implementation/API to KDE2 will change in some way or not. Yet another factor is that there are absolutely no available documentation for the different technologies, just the present API reference. To get the information on how the technology is supposed to work the existing source codes has to be analyzed.

### 4.4.1. KOM/OpenParts

The KOM/OpenParts technology is based on the open industry standard for distributed technology, CORBA 2.0. Around CORBA, KDE has developed a layer called KOM. KOM adds functionality to CORBA that is not provided by the CORBA Standard and specific to the application of distributed object technology to application framework development. KOM stands for KDE Object Model. The KOM Plug-ins can be implemented as in process (shared librar-

ies) or out of process servers (separate processes). There are some additional layers to the KOM layer to make it more user friendly to use the distributed technology, namely OpenPart-Controls and OpenPart-Part. [9]

*OpenParts -Controls*

This solution is for the modules, controls in KOM. Controls can be a complete module, a web browser with GUI, or just a simple function converting text. The control part supports controls to combine KDE components and X11 Windows. Controls can be implemented as in process or out of process servers. Controls can be swallowed in their parent window to extend embedded functionality. The controls are comparable to Microsoft's ActiveX Controls.

*OpenParts - Parts*

This solution is for the shell, the main application. The parts (controls) share limited resources such as Toolbars, Menubars, Statusbars etc. and they need a special toplevel window: a shell. The shell owns the File-Menu/Toolbar and the active part has access to the resources. Parts (Containers) can host other parts. Compare to Microsoft's OLE.

Figure 12 shows an abstract picture of how KDE/KOM OpenParts is constructed.

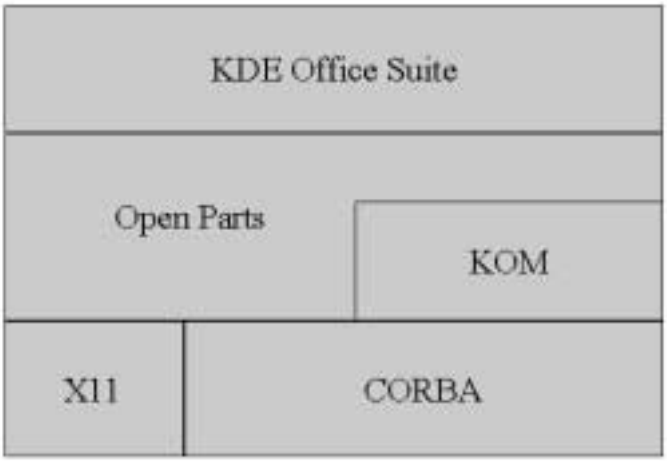


Figure 12: KDE KOM/OpenPart Technology

X11 and CORBA is the base in this technology. X11 is the graphical environment and the Mico ORB, the CORBA implementation used by KDE, implementing the distributed technol-

ogy. KOM and OpenParts provides a more user friendly API to the user (the application) who does not need to concern about complicated CORBA function calls.

KOM/OpenParts could be used to solve the dynamic extension of functionality problem by implementing a distributed solution, figure 6. KOM/OpenParts supports functionality to handle the repository. That is, functionality to register a new module, get the name and location of a module etc. Once a reference to a remote module is obtained it is implemented as it was a local module. This technology also handles the embedding of graphical components.

### 4.4.2. KParts

The KDE/KParts is a library that provides a framework for applications that want to use parts (the loadable modules in KParts). This technology has a lot in common with the KOM/OpenParts technology, except that is not distributed. The parts in KParts is similar to the KOM/OpenParts technology, they can be anything from a complete web browser with a GUI to a small function executing a calculation. [10]

The main applications need to inherit the main window from `KParts::MainWindow` and provide a so-called shell GUI, which provides a basic skeleton GUI with part-independent functionality/actions. That is to make the shell able to provide the functionality to dynamically locate, load and show parts.

The parts, the modules to be embedded in the shell, has to implement a given framework to be embeddable. KParts applications will not be specific to a given part, it has the functionality to extend the application and to embed any part, for instance, any viewer. For this the basic functionality of any viewer has been implemented in `KParts::ReadOnlyPart`, which viewer-like parts should inherit from. The same applies to `KParts::ReadWritePart`, which is for editor-like parts.

It is possible to add actions to an existing KParts application from the “outside”, defining the code for those actions in a shared library. This mechanism is obviously called plugins, and implemented by `KParts::Plugin`.

KParts could be used to solve the dynamic extension of functionality problem by creating a main application. This has to inherit and implement the `KParts::MainWindow` class. The

parts has to inherit and implement either the `KParts::ReadOnlyPart` class or the `KParts::ReadWritePart`.

### 4.4.3. General Aspects

KDE is a complete window manager using the Qt widget library. Qt is developed and supported by Troll Tech AS located in Norway.

*Technical aspects:*

KDE is a Desktop Environment for any Unix platform. While it is true that most KDE developers use Linux, KDE runs on a wide range of systems. There might be some problems to compile on some systems, and the source code may have to be altered a bit to get KDE to compile on a not so popular variant of Unix, or if the GNU development tools is not used, in particular the gcc compiler. [11]

Some of systems on which KDE is running are:

- Linux
- Solaris
- FreeBSD
- IRIX
- HP-UX

*Distribution and licence agreement:*

KDE is an Internet project. Development takes place on the Internet and is discussed on mailing lists and USENET news groups. No single group, company or organization controls the KDE sources. All KDE sources are open to everyone and may be distributed and modified by anyone subject to the well known GNU licenses. [8]

KDE is free software according to the GNU General Public License. All KDE libraries are available under the LGPL making commercial software development for the KDE desktop possible, all KDE applications are licensed under the GPL.

That is, KDE can be used to write libraries for “commercial and closed source” as well as “commercial and open source” software. If open source software is written then the Qt free edition may be used. But if closed source software is written the Qt free edition may not be

used. In the case of closed source software the Qt professional edition has to be obtained from Troll Tech AS [12].

## 4.5. GNOME

GNOME is the GNU Network Object Model Environment. The GNOME project intends to build a complete, easy to use desktop environment for the user, and an application framework for the software developer. GNOME is part of the GNU project, and is free software compliant with the OpenSource definition. [13]

GNOME provides a framework for building applications by providing a set of core libraries. These include libraries to create graphical user interfaces, components for creating applications with a uniform look and feel, and a CORBA ORB implementation named ORBit. [14]

The widget toolkit that GNOME use, GTK+, is written primary in C, although a large number of language bindings are available. Since GTK+ is implemented in C it is not as object oriented as KDE which uses C++.

The GNOME window environment provides a few technical solutions to get a component structured model. The basic facility is the GNORBA, GNome cORBA framework, that allows applications to use the GNOME implementation of CORBA, ORBit. Another facility provided by GNOME to write reusable software components is the Bonobo. Bonobo components are pieces of software that provide a well defined interface and are designed to be used in cooperation with other components. Using Bonobo makes it possible for GNOME applications to embed graphic and functionality supplied by other applications, compare to KDE KPart. CORBA is used as the communication layer that binds Bonobo components together, making it possible to distribute components over a network.

### 4.5.1. The GNOME CORBA Framework

The GNOME CORBA framework allows applications to use ORBit, the CORBA implementation used by GNOME.

To allow applications to request access to a specific CORBA object, GNOME CORBA servers place information in the repository named GOAD, GNOME Object Activation Directory, in

GNOME. The GOAD stores information on the CORBA objects that a program can provide to other programs. Each entry contains a unique implementation identifier (the “GOAD ID”), a list of interfaces that the object supports and information on how to create a new instance of the object implementation.

If an application provides the implementation for a CORBA object, it is necessary to integrate that object into the GOAD. An application would install a ’.goad’ data file into the correct directory as part of its installation process. Then a few function calls must be made when the object is created and destroyed. Once an object implementation is registered with GOAD, client applications can activate that implementation with a single function call. [15]

GNORBA could be used to solve the dynamic extension of functionality problem by implementing a distributed solution, figure 6. GNORBA supports functionality to handle the GOAD. That is, functionality to register a new module, get the name and location of a module etc. Once a reference to a remote module is obtained it is implemented as it was a local module.

### 4.5.2. Bonobo

Bonobo is a set of CORBA interfaces that define the interactions required for writing components. Bonobo is the architecture that makes components available to other applications as a Bonobo component. This enables applications to be embedded into another application for editing or displaying information. Bonobo makes GNORBA more user friendly by providing wrapper functionality for it. [16]

Bonobo can be used to solve both the dynamic extension of functionality and the comprehensive window control. By constructing the modules as a Bonobo component, the component provides new functionality to the Control Application. Since a Bonobo components also supports functionality to be graphically embedded a module can provide its own GUI that is presented by the Control Application.

### 4.5.3. General Aspects

GNOME is not a window manager and is not tied to any one window manager. GNOME is the GNU Network Object Model Environment. The GNOME project intends to build a complete desktop environment for the user.

*Technical aspects:*

GNOME was started by several people well-known in the Linux and GNU communities, but it is intended to run on any modern and functional Unix-like system. GNOME has been reported to work under the following [17]:

- GNU/Linux
- BSD (FreeBSD, NetBSD and OpenBSD)
- Solaris
- IRIX
- HP-UX
- AIX

What are the System Requirements for GNOME?

Currently, a machine with Unix or a Unix-like operating system installed is needed, with the X Window System (X11R5 or later). GNOME needs at least 16MB of RAM, although 32MB or more is recommended.

*Distribution and licence agreement:*

The widget toolkit that GNOME use, GTK+, is licensed under the LGPL. Like KDE GNOME is an Internet project. All sources are open to everyone and may be distributed and modified by anyone.

## 4.6. Summary

All the technologies discussed offers one or several ways to solve the problems of dynamic extension of functionality and comprehensive window control in SEA. Each technology provides two main solutions, one distributed and one that is not.

The distributed solution in general makes it possible to have instances of modules anywhere on a network. A disadvantage is that the degree of complexity increases using distributed technologies. All the distributed technologies provide functionality to get the name and location of modules to be loaded, this information is stored in the repository, figure 6. Tcl-DP provides RPC functionality, and does not support functionality to locate distributed modules.

Using a non distributed solution is a lot simpler at cost of flexibility. A problem that has to be solved in a non distributed solution is were to find the modules, a couple of solutions are possible, namely: Increase the functionality of each module in SEA or to have the information in a separate GUI configuration file.

Table 2 is a summary of the analyzed technologies

Table 2: Summary of technologies

| Technology | DEoF[a] Distributed | DEoF Non Distributed | CWC[b] |
|---|---|---|---|
| Tcl/Tk | Tcl-DP | Tcl source command | Passing widget paths as argument |
| Java | RMI / CORBA | Dynamic class loading | Java panels |
| KDE2 | KOM/OpenParts | KParts | KParts |
| GNOME | Bonobo | - | Bonobo |

a. DEoF: Dynamic Extension of Functionality
b. Comprehensive Window Control

## 4.7. Selected Technologies

All the distributed solutions will work, but they will all give a overhead and a high level of complexity to the application that is not needed. So the selected technologies are all non distributed alternatives. The selected technologies to implement are:

- Tcl/Tk using the source command and namespaces.
  This solution is selected since it is the simplest way to modify the existing code EIN/TSP already have.
- Java using dynamic class loading.
  Java is selected because is a popular object oriented language well suited for graphical development.
- KDE2 using the KPart technology.
  KDE2 is selected because its KParts technology solves both the problems of dynamic extension of functionality and the comprehensive window control.

The reason why GNOME is not one of the selected technologies to implement is that there is not enough time to make a test implementations using all technologies listed in Table 2. Since KDE2 and GNOME are both desktop environments the authors decided to only implement one of them. The reason why KDE2 is selected over GNOME is mainly because of KDE2's KPart that solves both the dynamic extension of functionality and the comprehensive window control in a non distributed way.

# 5. Implementation

This part describes the implementation of the selected technologies. First the Tcl/Tk implementations is described, second the Java implementation and finally the KDE2 implementation is described. Each technology describes how to solve the earlier stated problems with dynamic extension of functionality and comprehensive window control.

## 5.1. Precondition

The following implementations does not initiate or start the SEA core, it just uses the MPH library to communicate with a already running SEA core. That is, first the SEA core application has to be started and initiated according to given configuration file. Once the SEA is up and running the main application in question can be started.

The different implementations are examples of how the component based GUI application could be implemented in the technology in question. The implemented test application does not support any functionality like the SEA Control Center as shown in figure 2. The purpose is just to show the possibilities to extend the Main Control Application with modules that are unknown at buildtime for the Main Control Application.

Each module in the Control Application can only communicate with one given component instance in the SEA core. That is a given graphical module can not communicate with more than one SEA core component instance to show their status in one and the same window.

## 5.2. Tcl/Tk

The Tcl/Tk application consists of a main Tcl/Tk application that handles the window control and modules that implements the functionality for controlling the separate components in the SEA core.

### 5.2.1. Precondition

This solution uses a separate configuration file to tell what GUI modules that corresponds to a given instantiated component in the SEA core.

The notebook widget is a part of the extension package BWidget, which therefore must be included in the Tcl/Tk application.

### 5.2.2. Solution for the Main Application

The description of the solution for the main application is divided into three parts. First the start of the Tcl/Tk application, second the instantiation of a selected component in the SEA core and third, a few other supported functionalities.

*Start of the GUI Application*

The Tcl/Tk application is started with two parameters, the first is the name of the machine on which the SEA core is running, the second parameter is the port to connect to. These parameters can be obtained from the status bar in The Sea Control Center, figure 2. In the figure 2 example the name of the machine on which the SEA core is running is ‘solstal’ and the port for the MPH is ‘51765’. First the connection to the SEA core is set up, the MPH connection, using the `MPH::OpenConnection` procedure. Next the Tcl/Tk application asks the SEA core, through the MPH socket connection, what instances it has using the `MPH::SearchByName` procedure. The procedure call returns a list of all instances in the SEA core.

The returned list of all instances is iterated to see if a given instance of a component has a corresponding graphical module associated to it or not, according to the GUI-configuration-file, as illustrated in figure 13. That is, the Tcl/Tk application receives a list containing for example comp1.inst1, comp1.inst2, comp3.inst1 and comp4.inst1. The actual name of the component will be like LI-1, LI-2 etc. where LI is the name of the component and the number is the instance of the component.

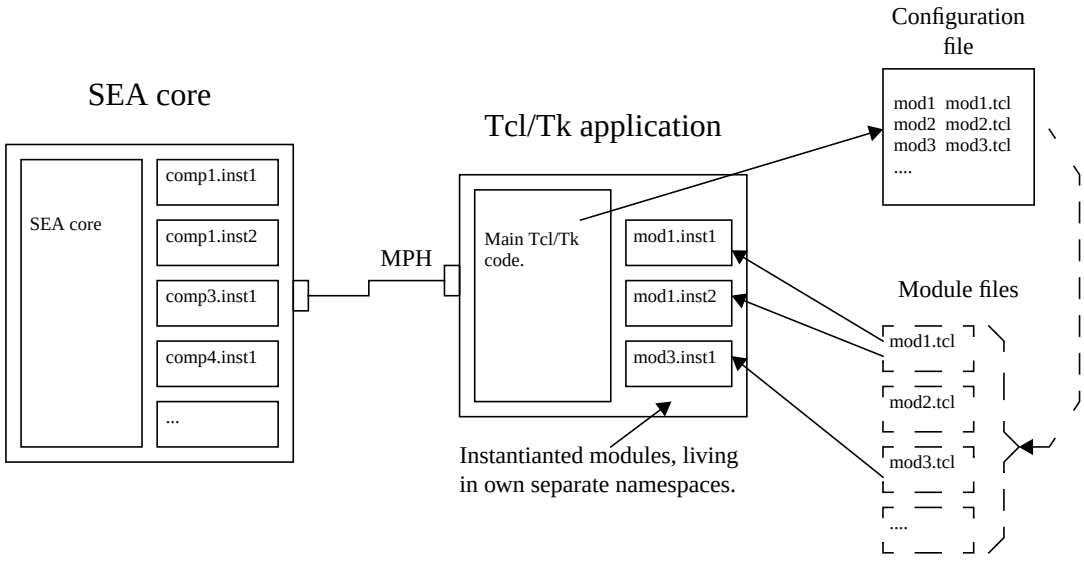


Figure 13: Tcl source and namespace commands

Once the list is obtained it will be iterated to see if there is any corresponding graphical modules according to the GUI configuration file that are associated to a given component type. If there is a corresponding module the component name and instance number, the identifier name, will be added to a list of available GUIs. The list of available modules may be graphically represented in a number of ways, in this solution it is shown as cascading menus with module type in the main menu and all the instances of each module in a submenu. The cascading menus are shown in figure 14.

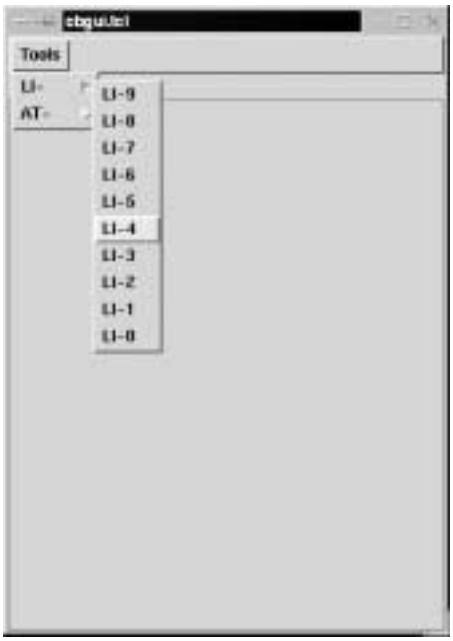


Figure 14: Tcl/Tk Control Application with cascading menus

*Instantiation of a Module*

The second step is to instantiate a module. When a instance of a given component is selected from the cascading menus, the `newEntry` procedure will be executed and the corresponding module to the selected component will be instantiated. First a new namespace is created using the Tcl command `namespace`, the new namespace will have the same name as the identifier. In the new namespace the module will be sourced using the Tcl command `source`, that is loaded into the existing Tcl/Tk code.

The module is now instantiated. Next step is to set up a MPH connection between the new module and the corresponding component in the SEA core. To do this a unique identification number has to be obtained from the module using the procedure `getIID`, which is a proce-

dure that all modules have to support. The `getIID` procedure returns the unique identification number.

Next step is to open a new MPH virtual socket using the `MPH::OpenChannel` procedure. The channel is a virtual socket over the MPH socket connection. This procedure is called with a number of parameters. One of the parameters is the IID for the module, telling the SEA core which component to connect to. If the procedure call `MPH::OpenChannel` was successful a new connection exists.

The last step in the instantiation scenario is to make an area for the module to draw its widgets in. In this Tcl/Tk solution the notebook[1] widget is used. A new tab is added to the notebook with the name of the identifier, module name and number. To create a new tab the `createTab` procedure is called. The return value of the call is the path to a frame widget within the tab, which should be used by the module to draw its graphics within. Finally the initialization procedure of the module is called, `proc init`, with the path where to draw its widgets passed as an argument.

At this point the module is instantiated, connected to the right component in the SEA core and the graphics of the module is drawn in a notebook tab widget.

*Other Functionality Supported*

The extra functionality supported by the main Tcl/Tk application is the ability to tear off a tab into a new toplevel window. That is, the main application supports this functionality for each separate tab that is created by adding an extra button to the tab. This tear-off functionality has to be done by hand in Tcl/Tk, in other languages there may be a direct support for tearing off graphical parts into own top level windows. Figure 15 shows the main Tcl/Tk application with a number of LI-telephone modules embedded and one torn off.

1. The notebook widget is a part of the extension package BWidget, which therefore must be included.

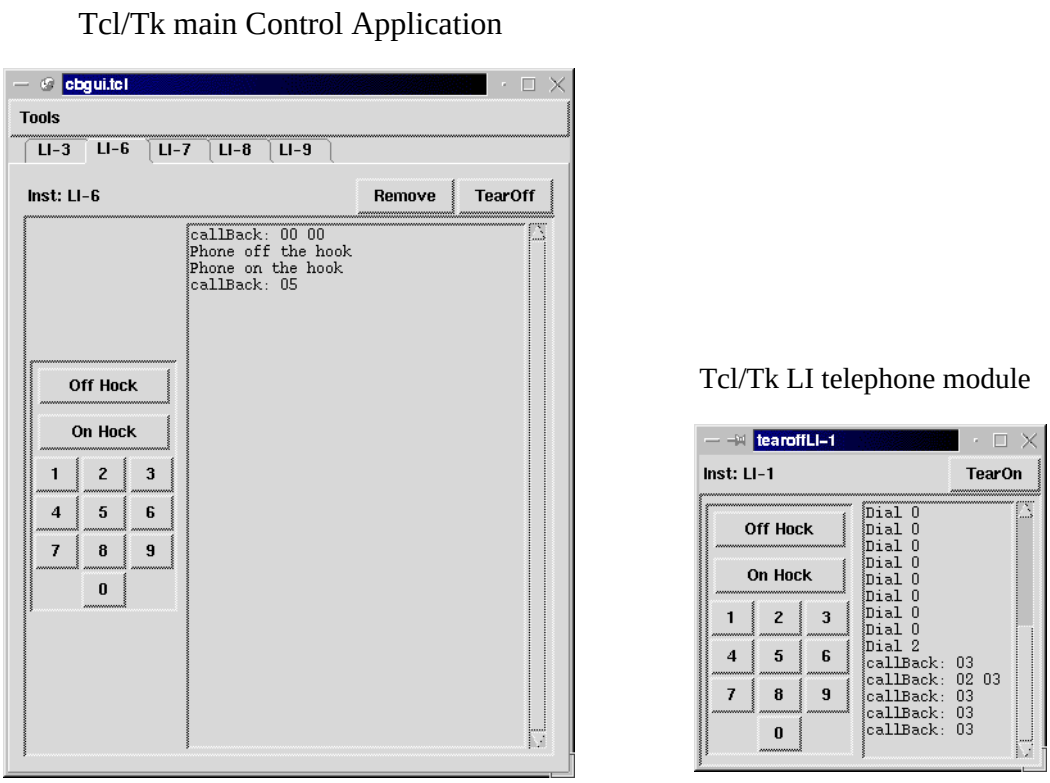


Figure 15: Module torn off from the main Tcl/Tk application

Yet another functionality that can be supported but still not implemented is the functionality for the module to extend the main applications menubar. In other words, when a given module tab is activated the module should extend the main applications menubar with extra functionality supporting the activated module. This can be solved using the same approach as when the main application gives the path to draw the modules widgets within. In the case of menu extension the path where to extend the menu has to be passed as an argument to a standard procedure in the module, that extends the functionality of the menubar.

### 5.2.3. Solution for the Modules

The LI-telephone module is the only module that is implemented. The LI-telephone module represents the general structure for a module. The modules structure is in short, a number of standardized procedures to support initialization and communication with the main application and the corresponding component in SEA, then a number of procedures to support the modules functionality.

There are some standard procedures that is needed for the module to set up the MPH connection and some standard procedures needed to create the GUI for the module, like the `proc`

`init` to initialize the GUI. The standard procedures can be compared to a pure virtual class in C++ since they have to be implemented. The rest of the procedures have specific functionality for the given module.

### 5.2.4. Implementation

The Tcl/Tk implementation consists of two separate files types, cbgui.tcl which is the main application in Tcl and the modules modX.tcl. The modules modX.tcl is only implemented for the LI-telephone module, li_gui.tcl.

#### 5.2.4.1. The cbgui.tcl file

Implementation file for the main application. The file is divided into three parts, procedures for the MPH communication, procedures for GUI functionality and finally general procedures for adding and removing new modules.

MPH procedures

- `proc SendMessage {channel message}`
  This procedure is used by the GUI instances to send text messages to its SEA entity. The parameters are `channel` which is the channel given to the instance and `message` is the text message to be written. The procedure does not return anything.
- `proc SendBinaryMessage {channel length message}`
  This procedure is used by the GUI instances to send binary messages to its SEA entity. The parameters are `channel` which is the channel given to the instance, `message` which is the text message to be sent and the `length` which is the length of the message. The procedure does not return anything.
- `proc concloseport {}`
  This is a callback procedure required by MPH, it is called if the MPH connection is closed. The procedure does not take any arguments and does not return anything.

GUI procedures

- `proc draw {}`
  This procedure creates and places the widgets, menubar and notebook, on the main window. The procedure does not take any arguments and does not return anything.
- `proc addToMenu {cascade name command}`
  This procedure adds a new command to the given cascade menu in the menu pointed out by

the global variable `menuPath`. If the cascade do not exist it is created and placed on the main window. The parameters are `cascade` which is the name of the cascading menu, `name` which is the name of the new command and `command` which is the code to be executed when the item is selected from the menu. The procedure does not return anything.

- `proc killTab {inst}`

  This procedure deletes a tab from the notebook. The parameter `inst` is the name of the tab to delete. The procedure does not return anything.

- `proc tearOff {inst channel mod}`

  This procedure makes a new toplevel window, deletes the specified instance and adds it to the new toplevel window. The parameters are `inst` which is the name of the instance to be in a new top level window (the tab to tear off), `channel` which is the MPH channel given to the module for communication with its component and `mod` which is type of instance. The procedure does not return anything.

- `proc tearOn {inst channel mod}`

  This is the reverse procedure of `tearOff`. It destroys a given toplevel window and creates a new tab in the notebook and initiates the module there. The parameters are `inst` which is the name of the toplevel to be placed in a new tab, `channel` which is the MPH channel given to the module for communication with its component and `mod` which is the type of instance. The procedure does not return anything. Note, the name `tearOn` was a joke at the beginning, but since we did not come up with a better name it remained this way.

- `proc initTearOffPage {page inst channel mod}`

  This procedure initiates the new toplevel window and creates a frame for the module to draw its widgets in. The parameters are `page` which is the new toplevel window, `inst` which is the name of the instance, `channel` which is the MPH channel given to the module for communication with its component and `mod` which is type of instance. The procedure returns the path to the drawing area for the module.

- `proc initPage {mod page inst channel}`

  This procedure is called immediately after a new tab has been created. It creates widgets for some standard functionality and the frame that is by sent to the GUI instance. The parameters are `page` which is the new page (tab), `inst` which is the name of the instance, `channel` which is the MPH channel given to the module for communication with its component and `mod` which is type of instance. The procedure returns the path to the drawing area for the module.

- `proc createTab {mod inst channel}`

  This procedure is called immediately after a instance has been selected from the menu. It creates a new tab and call `initPage` to initiate it. The parameters are `inst` which is the name of the instance, `channel` which is the MPH channel given to the module for communication with its component and `mod` which is type of instance. The procedure returns the path to the drawing area for the module.

General procedures

- `proc newEntry {mod inst tclappsourc}`

  This procedure is called if the user selects a entity from the menu. It loads the source code for the given instance. The parameters are `mod` which is the type of instance, `inst` which is the name of the instance and `tclappsource` which is the file containing the new source code to load. The procedure does not return anything.

`proc destroyEntry {mod inst channel}`

This procedure removes a GUI instance by disconnecting its MPH channel, enable it in the menu and remove its tab. The parameters are `mod` which is the type of instance, `inst` which is the name of the instance and `channel` which is the MPH channel given to the module. The procedure does not return anything.

- `proc addEntities {}`

  This procedure checks available entities in SEA against the configuration file ’cbgui.cfg’ and adds the matching entities. The procedure does not take any arguments and does not return anything.

- `main`

  main is not a real procedure in Tcl/Tk, it is the global code in the file which acts like the main code. All the necessary calls and initiations are done form here. Note all the variables in a module that are declared here will belong to the global namespace.

#### 5.2.4.2. The li_gui.tcl

Implementation file for the LI module, representing any module. The file is divided into two parts, general procedures that has to be supported to interact with the main application and procedures specific to the module.

General procedures needed to interact with the main application.

- `proc getIID {}`

  This procedure returns the unique IID number for the specific module. The procedure does not take any arguments.
- `proc messagecallback {length message}`

  This procedure is called when a message has been received. The parameters are `length` which is the length of the message and `message` which is the actual message. The procedure does not return anything.
- `proc messagecallback {}`

  This procedure is called if the channel is remotely closed by the SEA component. The procedure does not take any argument and does not return anything.
- `proc init {_path _channe}`

  This procedure is called when the module is instantiated. The parameters are `_path` which is the path to draw the modules widgets in and `_channel` which is the MPH channel to communicate with the corresponding component. The procedure does not return anything.

Specific functionality for the given module. In this case the LI-telephone module.

- `proc offHook {}`

  This procedure sends a offHook message to its SEA component. The procedure does not take any argument and does not return anything.
- `proc onHook {}`

  This procedure sends a onHook message to its SEA component. The procedure does not take any argument and does not return anything.
- `proc dial {digit}`

  This procedure sends a selected digit to its SEA component. The parameter `digit` is the digit to dial. The procedure does not return anything.
- `proc setInfotext {text}`

  This procedure prints text in the textarea. The parameter `text` is the text to print. The procedure does not return anything.

Note in the module there can not be any “main” code, this code will never be executed.

### 5.2.5. Conclusions

Using Tcl/Tk is a simple way to solve this problem in a small scale application like this example. In a far more complex structure like in the SEA it has to be well designed with hard specifications on naming conventions and functionality that each standard function should perform. If Tcl/Tk is to be used in a large scale application it would be a good idea to look at some class abstraction package to extend Tcl.

## 5.3. Java

The Java implementation consists of the main application that handles the window control and the code modules that implements the functionality for controlling the separate components in the SEA core. The code for controlling the components is implemented as separate Java classes (module classes). Both the main application and the module classes uses the already implemented MPH library for Java to communicate with SEA. In this implementation the Java Swing classes is used for the GUI. The main application is implemented as a Java applet.

### 5.3.1. Preconditions

In the Tcl and C versions of the MPH library there exists functionality to get the name of all the currently loaded components in SEA, but not in the Java version. This functionality must be added to the Java MPH library in order for this implementation to work in reality. To get around this problem in this test implementation a few known component names are explicitly declared in the code.

### 5.3.2. Solution for the Main Application

The main application consists of a menu bar (JMenuBar) and a tabbed pane (JTabbedPane). The menu bar has one menu (JMenu), named “tools”. The tools menu contains the name of the components that can be selected. Each pane in the tabbed pane contains a panel (JPanel) that is used by the module classes, one panel for each module class.

To start the application two parameters are needed, the host and port for the running SEA. In this test application those parameters is given in the html file that is used to start the applet. When the Java application starts it creates a object of the MPHclient class, this object contains the methods that is used to communicate with the SEA core. Now the application should get the names of the loaded components in SEA and compare these to a configuration file in order to know which components that have an associated module class and add these to the menu. This is not possible, see the precondition for Java, so instead a few known component names

are added to the menu. Finally an instance of class `Tabs` that inherits from a `JTabbedPane` is created and added to the main application window. Figure 16 shows the main application with the cascading menu.

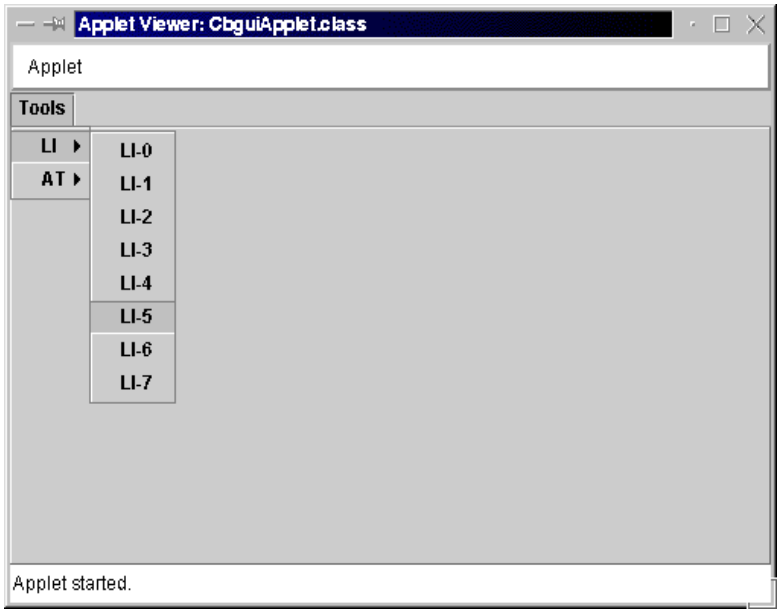


Figure 16: The Java without any module classes loaded.

When the user selects a component name from the menu a callback method is called with the name of the component as a parameter. This name is the same as the class that is to be loaded, for example, in SEA the LIC components are called LI-1, LI-2,.... so the module class to load when the user has selected a LIC module is called LI.class. Then the given class is loaded into the Java interpreter with the static method `Class.forName()` and instantiated with the method `newInstance()` in the `Class` object returned by `forName()`. After the module class has been instantiated it has to be initiated. The service provided by the main application to the module classes are a MPH connection to the SEA component associated with the given module class and a `JPanel` that is used by the module class for all its user interaction. To set up a connection to a SEA component the main application calls the method `connect` in the `MPHclient` object, this method returns an instance of the class `MPHconnection`. The `MPHconnection` class contains methods used for sending messages to SEA. A `JPanel` is created and added to a new tab in the `JTabbedPane`. The `MPHconnection` and the `JPanel` are passed as arguments to the method `init()` in the module class.

### 5.3.3. Solution for the Module Classes

The module classes is where the functionality for controlling the different components are implemented. In order for this classes to work with the main application and the MPH library they must implement two interfaces, `MPHclientListener` and the `CompInterface`. The `MPHclientListener` interface contains method declarations that is used to receive messages from the SEA component. The `CompInterface` contains methods used by the main application to initiate the Module Class.

When a component is selected in the main application an instance of the given module class is created. After the creation of the Module Class object it is initiated by a call to the method `compInit` with the `MPHconnection` and a `JPanel` as parameters.

In order for the Module Class to interact with the user of the Control Application it must use the given `JPanel`. The placement of this panel is controlled by the main application. In this implementation the `JPanel` is placed on a tab in the `JTabbedPane` in the main application but could just as well be placed in its own toplevel window.

For the communication between the Module Class and its SEA component the `MPHconnection` object and the methods defined in the `MPHclientListener` interface is used. The `MPHconnection` object contains the method `send()` that the Module class use to send MPH messages to its SEA component. The `MPHclientListener` interface contains declarations for callback methods used by SEA to send MPH messages back its client. Since the Module Class must implement the `MPHclientListener` it must implement these methods.

### 5.3.4. Implementation

The implementation is divided into two implementation parts. First, the implementation for the Main Application is described and secondly the implementation of one Module Class, the LIC component, is described.

#### 5.3.4.1. The Main Application

The main application consists of three classes, CbguiApplet, Tabs and ComponentList. CbguiApplet is the main class, Tabs is used to create and handle the tabs in the tabbed pane and ComponentList is the class that handles the representation of the available module classes. Figure 17 shows small design pattern for the Java implementation drawn in Booch notation.

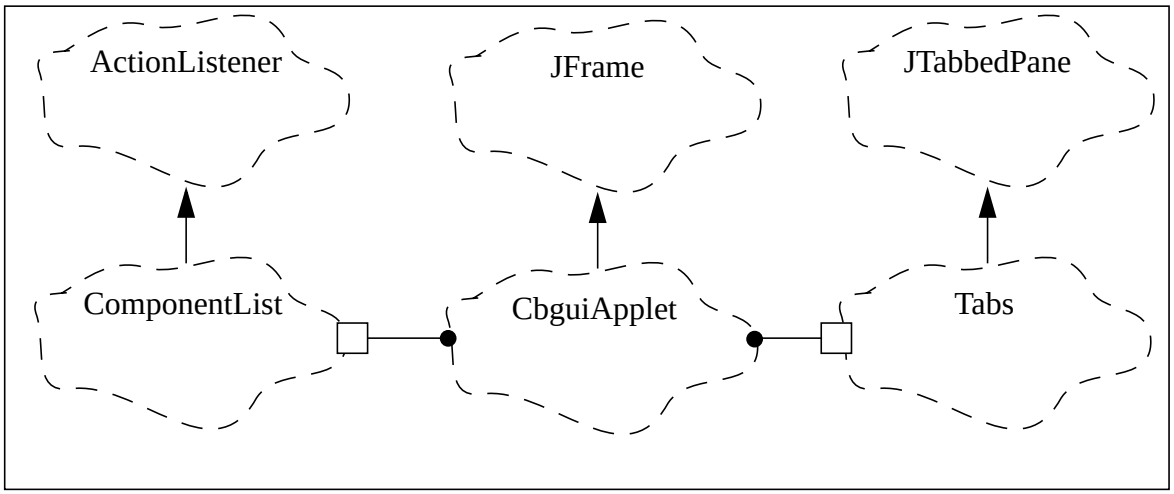


Figure 17: Structure of the Main Application in Java.

*public class CbguiApplet extends JApplet:*

This is the applet class which is executed from the html file.

Class members:

- `MPHclient mph`

  The MPHclient object contains the methods to communicate with SEA core. It gets instantiated in the constructor.
- `ComponentList compList = new ComponentList(this).`

  This object handles the menu.
- `Tabs tabs = new Tabs(this).`

  This object handles the TabbedPane.
- `public init()`

  This method is called when the applet is created, it gets the host and port parameters from the html file and establish a new connection to SEA by creating an instance of MPHclient.
- `public void addComponents()`

  This method is supposed to retrieve the loaded SEA components and the available module classes and adding those to the component list, the menu, by calling the newEntry in the ComponentList object. However, this can not be done due to the problem with the MPH library described in the Java precondition. Instead a few known component names are added to the menu.

- `public void itemSelectCallback(String inst)`
  This is the callback method that is called when the user selects an item in the component list. It loads and initiates the module class for the selected item. A new JPanel is created and added to the TabbedPane by calling addComponent in the Tabs object. A new instance of the module class is created by first loading it into the Java interpreter with Class.forName and then instantiate it with the method newInstance in the Class object returned by Class.forName. A MPH channel to the given SEA component is created with the method connect in the MPHclient object, this method returns an instance of MPHconnection. Finally the method init in the module class object is called with the newly created MPHconnection and the JPanel as parameters.
- `public void destroy()`
  This method is called when the applet is destroyed. Is closes the MPH connection by calling the closeConnection method in the MPHclientListener class.

*class Tabs extends JTabbedPane*:
This class handles the TabbedPane and the JPanels that the Module Classes use as their drawing areas.

Class members:
- `public Tabs(JFrame parent)`
  Adds the Tabbed pane to the given JFrame.
- `public JPanel addComponent(String inst)`
  Creates a new JPanel and adds a new tab to the tabbed pane. Then adds the JPanel to the new tab and returns the JPanel.

*Class ComponentList implements ActionListener*:
This class handles the menu representation of the available module classes.

Class members:
- `JMenuBar menuBar`
  The menu bar.
- `JMenu menu`
  The “Tools” menu.

- `CbguiApplet owner`
  The instance that created this object.
- `public ComponentList(CbguiApplet cb)`
  Constructor, creates and adds the menu bar and the menu to the application.
- `public void newEntry(String mod, String inst)`
  Adds a new item, “inst”, to the cascade menu “mod”. If “mod” does not exist it is created. Since the class ComponentList implements ActionListener the item is assigned this as the action listener.
- `public void actionPerformed(ActionEvent e)`
  Method defined in the interface ActionListener. It gets called when the menu item is selected. It calls the itemSelectCallback() method in the Cbgui class with the instance name as a parameter.

#### 5.3.4.2. The Module Class

Every module class must implement two interfaces, the `MPHclientListener` interface and the `CompInterface` interface.

*MPHclientListener*

The MPHclientListener defines the callback methods used by SEA to send MPH messages to the client.

Members:

- `public void receiveMessage(byte[] data, int length)`
  It is this method that receives the messages to the Module Class. The message comes as a byte array, it is up to the implementation of the Module Class to convert the message to something that can be understood by the receiver.
- `public void connectionClosed()`
  This method is called if the Main Application loses its connection with SEA, which means that the Module Class MPH channel also gets closed.
- `public void channelClosed()`
  This method is called if the MPH channel is closed.
- `public void error(String errMsg)`
  This method is called if an error occurs in the MPH communication.

*CompInterface*

The CompInterface defines the methods used by the Main Application to initiate and embed the GUI of the Module Class.

Members:

- `public void init(MPHconnection mph, JPanel p)`
  This method is used to initiate the Module Class. The parameters is the MPHconnection used to communicate with SEA. The JPanel which is the drawing area that this Module Class must use for all user interactions.
- `public String getIID()`
  Every component in SEA has an identifier id. This method must return the identifier for its SEA component.

*public class LI implements CompInterface, MPHclientListener, ActionListener*

This class is a simple Module Class implementation that communicate with a LIC component in SEA. It is dynamically loaded and instantiated by the CbguiApplet. The GUI consist of a simple keypad and a text field. Figure 18 shows the main application with three LIC modules loaded.

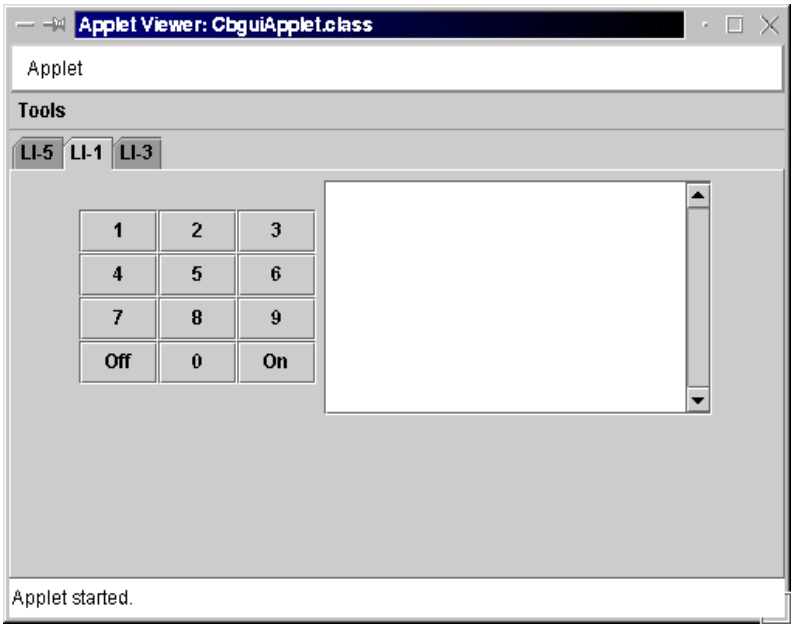


Figure 18: The Java application with three LI modules loaded.

Class members:

- `MPHconnection mphConn`

  Contains methods for sending messages to the given LIC component.
- `JTextArea text`

  The text area.
- `public String compGetIID()`

  Method required by the CompInterface interface, it returns the identifier ID for the LIC component.
- `public void compInit(MPHconnection mph, JPanel p)`

  Method required by the CompInterface interface, this method is called right after the object has been created. In this method the GUI is created and added to the given JPanel. The parameters is the MPHconnection and the JPanel used by this class.
- `public void receiveMessage(byte[] data, int length)`

  Callback method required by the MPHclientListener interface. This method is called when a message from the given SEA component is received by the MPH.
- `public void connectionClosed()`

  Method required by the MPHclientListener interface, it does not do anything.
- `public void channelClosed()`

  Method required by the MPHclientListener interface, it does not do anything.
- `public void error(String errMsg)`

  Method required by the MPHclientListener interface, it does not do anything.
- `public void actionPerformed(ActionEvent e)`

  Method required by the ActionListener interface. Since the buttons on the keypad in the GUI uses this as actionlistener this method define the action for the buttons.

### 5.3.5. Conclusion

Using Java would be a simple way to implement the Control Application. Java is object oriented and the problem to solve is suited for object orientation. It is also simple to handle GUI in Java. This makes Java suitable for the implementation of the Control Application. The problem is that the Java MPH library does not include the same functionality as the Tcl and C versions of it, se the Java preconditions.

## 5.4. KDE2 KParts

We intended to make an implementation using KParts, but there was a number of circumstances that made it rather complicated. One of the main problems was that KDE2 is still in alpha release and there were great difficulties to build the KDE2 source code for the Sun Solaris platform. After approximately two weeks of non successful compilation of the KDE2 source code for the Solaris platform, it was given up[1]. Instead we tried to build the KDE2 source code for the Linux platform on an Intel x86. This was eventually successful. Since KDE2 is not officially released there exists absolutely no documentation for the different parts and technologies supported by KDE2, like the KParts technology. To get the necessary information the source code for the different parts has to be analyzed. Due to lack of time we have not been able to make a complete KPart implementation, so this section only describes the idea of the KPart technology and how it could be used to implement a SEA Control Application.

### 5.4.1. The KParts Technology

The Kparts technology supports solutions for both dynamic loading and graphical embedding of new modules. KParts is a C++ library that consists of several classes to provide a framework for development of both the main application, a shell in KParts, and the modules, parts in KParts.

All the KParts classes are encapsulated in a namespace, KParts. In other words, when instantiating a KParts class the syntax is KParts::ClassName().

A KParts part is a GUI component, featuring a widget embeddable in any shell application. If a part does not support editing functionality, a “viewer”, it must inherit and implement the `KParts::ReadOnlyPart` class. If the part is both viewable and editable, an “editor”, it must inherit and implement the `KParts::ReadWritePart` class.

The shell is a KPart-aware main window, that is a window that can embed the GUI of a dynamically loaded class. A shell application has to inherit the `KParts::MainWindow` class in order to embed KParts parts.

1. The KDE2 is in alpha release and consists of hundreds of megabytes of source code. Compiling a stable version, which means no compilation errors, takes approximately one day for all parts.

In order for the shell to be able to load and instantiate a given part, the part has to be compiled as a shared library. The shared library must contain a initialization function, a factory class and the implementation of the part as a separate class.

The initialization function must follow a specified naming convention, for example the part notepad's library name must be *libnotepad.la* and the initialization function must be named `init_libnotepad()`, as shown in figure 19.

```
extern "C"
{
    void* init_libnotepad()
    {
        return new NotepadFactory;
    }
};
```

Figure 19: KPart initialization function.

The initialization function is declared as extern C so that the compiler does not change the function name, which it does for ordinary functions. This is needed due to the fact that the part is never linked to the shell application. It is only known by name and therefore the name of the initialization function must be standardized. The initialization function returns a static factory object for the given part.

The factory class must inherit and implement the `KLibFactory` and overload the functions `create` and `instance`. The KLibFactory implements the singleton pattern. The create function creates the part of the given type and returns a reference to it. Now the shell has a pointer to a instance of a part of any type.

The shell has GUI elements that the parts want to share and extend, like the menubar, toolbar and statusbar. The layout of these elements are described in a XML file for the shell. The parts also supports GUI elements that will be merged in the shell's user interface. The layout of the parts element interface in the GUI is also defined by an XML file for each part.

### 5.4.2. Implementation

Since the KParts technology supports both dynamic loading and graphical embedding of new modules it is possible to create a SEA Control Application using this technology. In practice a solution for this could be solved as illustrated in Booch notation in figure 20.

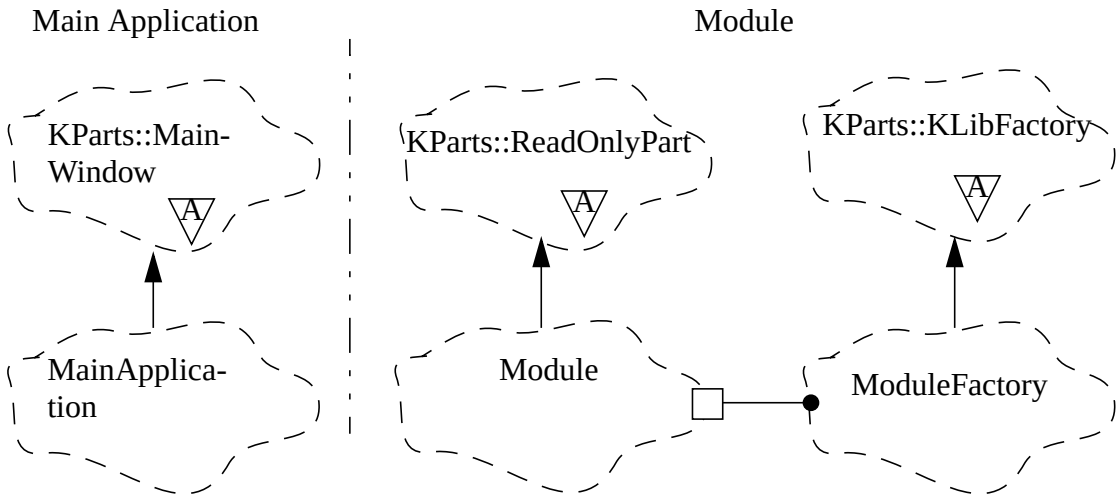


Figure 20: Structure of the Main Application using KParts.

The main application inherits and implements the `KParts::MainWindow` to be able to embed KPart modules. The modules, which must be compiled to a shared library, has to inherit and implement both the `KParts::MainWindow` class and the `KParts::MainWindow` class. The Main application has to use the MPH C library function `MPH_SearchByName()`, to get a list of all instances in the SEA Core. This list has to be analyzed to see if there exists a corresponding library file to the component. If there exists a module it will be added to a list of available modules, like the Tcl and Java solutions.

When a module is selected by the user to be shown, it has to be instantiated. For example let's say the user selects to show a LIC-telephone component. This is done in the following way. First the name of the module determines the name of the library to be loaded. In this case it would be *libli.la*. Then the `init_libli()` function is called and a factory for the LI class is returned. The LI factory object is then used to obtain the instance of a LI object using the `create` function in the LI factory. One of the arguments the create function takes is the name of the parent widget, this is for the module to know where to draw its widgets. The create function returns a pointer to the newly created LI instance. At this point the Main Application is extended with a LI module. When the LI module gets activated the li.rc file is executed. The

li.rc file contains the XML code for the LI module extend to the Main Application. In this way the Main Application updates the functionality for each activated module.

### 5.4.3. Conclusion

Even if we did not make an implementation of this technology, we still believe that when the KDE2 comes in a final and stable release this technology is an interesting candidate for the SEA Control Application. Hopefully the final release will contain a much more detailed documentation of the KParts.

# 6. Conclusion

All the analyzed technologies offers one or several ways to solve the problems of dynamic extension of functionality and comprehensive window control in SEA. The problems can be solved in a distributed and a non distributed way as described in chapter 3, general solution. Using a non distributed solution is a lot simpler than using a distributed, at cost of flexibility. We have chosen to focus on the non distributed solutions provided by Tcl/Tk, Java and KDE2/ KParts.

Using Tcl/Tk is a simple way to solve the problems of dynamic extension of functionality and the problem of comprehensive window control. Since Tcl/Tk is a interpreted language it is easy to extend its functionality at runtime. In a small scale application Tcl/Tk is easily managed and structured but in a large and complex application the structure might get a bit confusing.

Java is a widely known object oriented language that works with bytecode running on a virtual machine making it possible to run the same bytecode on different platforms. It is also possible to run the same code both as an applet and a standalone application. The Java swing classes makes it easy to create GUIs. The disadvantage of Java is that it is slower than for example Tcl. Both when running but especially at startup when the Java interpreter has to start.

KDE2/KParts technology offers a nice technical solution to solve the problems of dynamic extension of functionality and comprehensive window control. This is an interesting solution but will have to wait for the final KDE2 release to be a real candidate. The solution will most likely be a bit larger and more complex than both the Tcl/Tk and Java solutions. An advantage for KDE is that it is “pure” binary code and therefore is fast to execute. A disadvantage is that KDE only exists for the Unix community.

Finally the technology and language that we recommend to use for the development of a Component Based Graphical User Interface is:

- Java using dynamic class loading functionality
- Tcl/Tk using namespaces.

Tcl/Tk is chosen since it is a tool for fast development of graphical applications. It is well suited to extend the functionality of already existing applications, that is using the source command. The drawback of Tcl is the namespace functionality. Using this in a large scale application will probably be difficult and hard to organize. To solve this problem of abstraction we propose that some class abstraction package to extend the Tcl is analyzed.

Java is chosen because it is a object oriented language and the problems to solve is suited for object orientation. Using the dynamic class loading functionality in Java makes it easy to extend the functionality of an already existing application, thus solving the problem of dynamic extension of functionality in a object oriented way.

KDE/KParts is not selected because it will most likely be a bit larger and more complex than both the Tcl/Tk and Java solutions. This higher level of complexity is not needed in this kind of solution. This decision is not based in the fact that we did not manage to make an implementation using KParts.

# 7. References

This chapter includes all references used in this thesis. First all the references used in the text then some books that are frequently used throughout the time of the thesis and last some general World Wide Web links were a lot of information on the different subjects can be found.

## 7.1. Indexed References in the Thesis


[1] http://www.ericsson.se/pressroom/comp_newtw.shtml

[2] http://www.ericsson.se/infotech/company/

[3] http://bokhyllan.ks.ericsson.se/

[4] http://www.pconline.com/~erc/tcl.htm

[5] http://dev.scriptics.com/software/tcltk/license_terms.html

[6] http://java.sun.com/docs/books/tutorial/rmi/overview.html

[7] http://java.sun.com/nav/business/license-faq.html

[8] http://www.kde.org

[9] http://www.kde.org/whatiskde/openparts.html

[10] http://developer.kde.org/documentation/tutorials/components/index.html

[11] http://www.kde.org/documentation/faq/kdefaq-2.html#ss2.3

[12] http://www.trolltech.com

[13] http://www.gnome.org/

[14] http://developer.gnome.org/arch/

[15] http://developer.gnome.org/arch/component/gnorba.html

[16] http://developer.gnome.org/arch/component/bonobo.html

[17] http://www.gnome.org/gnomefaq/html/x131.html


## 7.2. General Book References


David Flanagan, *Java in a nutshell*, Second Edition, O'Reilly, 1997

John K. Osterhout, *Tcl and the Tk Toolkit*, Addisson Wesley, 1994

Grady Booch, *Object-Oriented analysis and design*, Second Edition, Addison Wesley, 1998

## 7.3. General URL References

*CORBA Technoligies*

| | |
|---|---|
| www.omg.org | General information about CORBA |

*Tcl/Tk*

| | |
|---|---|
| www.scriptics.com | The homepage of Tcl/Tk |
| www.sco.com/Technology/tcl | Miscellaneous information |
| www.tcltk.com | Miscellaneous information |

*Java*

| | |
|---|---|
| java.sun.com | The Source for Java Technology |

*KDE*

| | |
|---|---|
| www.kde.org | General information about KDE |
| developer.kde.org | KDE Developers' Web Site |
| www.mico.org | Mico, the CORBA implementation used by KDE |
| www.trolltech.com | Qt homepage |

*Gnome*

| | |
|---|---|
| www.gnome.org | The GNOME Project |
| developer.gnome.org | GNOME Developers' Web Site |
| orbitcpp.sourceforge.net | ORBit, the CORBA implementation used by GNOME |

*Licences*

| | |
|---|---|
| www.gnu.org/copyleft/gpl.html | GNU General Public License, GPL |
| www.gnu.org/copyleft/lgpl.html | GNU Lesser General Public License, L-GPL (formerly known as the GNU Library GPL) |
| www.opensource.org/osd.html | Open Source Licence |

# Appendix A: Abbreviations

List of abbreviations used in the thesis.

| | |
|---|---|
| API | Application Programmers Interface |
| CORBA | Common Object Request Broker Architecture |
| EIN | Ericsson Infotech AB. |
| GNU | GNU is Not Unix |
| GPL | GNU General Public License |
| GUI | Graphical User Interface |
| L-GPL | GNU Lesser General Public License |
| LIC | Line Interface Circuit |
| MPH | Message Protocol Handler.<br>Multiplexed TCP/IP socket connection protocol. |
| ORB | Object Request Broker (Often a specific implementation of the CORBA technology, “an ORB”) |
| SEA | Simulated Environment Architecture.<br>EIN/TSP’s simulator environment to simulate an AXE switch. |
| SS7 | Signaling System no.7 |
| TSP | Department of Test and Simulated Platform. |
| QPL | Q Public License |
| XML | Extended Markup Language |

# Appendix B: Description of the thesis

This appendix shows the official description of the thesis, stated by Ericsson Infotech AB.

C-DEGREE THESIS

COMPONENT BASED GRAPHICAL USER INTERFACE

## Students

Johan Torbjörnsson and Peter Svensson at Karlstads Universitet.

## Goal

The goal of the thesis work is to propose a suitable technology for designing component based user interfaces.

## Background

Today Ericsson Infotech (EIN) has a simulation product that is built using components. The components are combined at run-time to create a simulation of the system the user needs. The system is divided in a simulation part and a control part. The component system used only covers the simulation parts not the graphical user interface (GUI) used to control the system. This leads to problems with trying to keep the GUI updated with all the different simulation components designed by EIN and third party providers. To solve this problem EIN would like to have a GUI system that is extensible a run-time so that a component can consist of a simulation part and an optional GUI part that adds functionality to the GUI.

## Execution

Evaluate the existing technologies that can be used to build a GUI that is run-time extensible using some form of component structure. Describe the pros and cons of the different solutions. Propose one or two technologies that are suitable for EINs needs. Build simple prototypes using the selected technologies.

## Time estimate

The time for the thesis work is 20 weeks at 50%.

## Result

Report describing the available solutions. Prototypes demonstrating the capabilities of the two most promising solutions. A presentation of the work at EIN.

## Contacts at EIN

Magnus Einarsson 054 19 35 20

Magnus.Einarsson@ein.ericsson.se

# Appendix C: Tcl/Tk Syntax

This appendix gives a brief introduction in Tcl/Tk and its syntax. For more information please use the links for Tcl/Tk in the reference chapter 7. For Tcl syntax information se http://www.sun.com/960710/cover/tcl-syntax.html#syntax.

The first part of Tcl/Tk is the Tcl interpreter which is a library of C procedures that implements the Tcl commands and the grammar for the Tcl language. This means that Tcl has no fixed grammar that explains the entire language. Instead, Tcl is defined by the interpreter that parses single Tcl commands and the procedures that executes the commands. The interpreter and its substitution rules are fixed, but new commands can easily be added and existing ones can be modified and replaced. In Tcl assignments, procedure calls and features that control the program flow such as if and while are all implemented as commands, that is, they are not understood directly by the Tcl interpreter.

The other part of Tcl/Tk is Tk. Tk is a graphical user interface toolkit that makes it possible to create GUIs quickly. Tk extends the built-in Tcl commands with commands for creating and controlling graphical user interface elements called widgets. A widget can be a button, text window, scrollbar etc. Widgets are arranged hierarchically on screen with one toplevel window as ‘root’. The syntax for accessing the different widgets in the hierarchy use dots (.). Tk ships with all distributions of Tcl.

A Tcl script file can be executed just like a shell script via the `tclsh` (Tcl shell) or `wish` (windowing shell). The `tclsh` does not support the Tk extension.

As mentioned earlier the Tcl interpreter is implemented as a library of C procedures and it is easy to extend the Tcl and Tk functionality by writing new C procedures and incorporate these in the Tcl interpreter as new commands. Due to this fact there is a lot of packages of extended functionality available to Tcl/Tk. Two of those extensions are Tcl-Dp and BWidget. Tcl-Dp contains functionality for distributed programming and BWidget is an extension of Tk with a number of new widgets.

*Basic syntax*

Tcl scripts are made up of commands separated by new lines or semicolons. Commands all have the same basic form illustrated by the following example:

```
expr 20 + 10
```

This command computes the sum of 20 and 10 and returns the result, 30. Each Tcl command consists of one or more words separated by spaces. In this example there are four words: expr, 20, +, and 10. The first word is the name of a command and the other words are arguments to that command. All Tcl commands consist of words, but different commands treat their arguments differently. The expr command treats all of its arguments together as an arithmetic expression, computes the result of that expression, and returns the result as a string. In the expr command the division into words isn't significant: you could just as easily have invoked the same command as

```
cmd arg arg arg
```

A Tcl command is formed by words separated by white space. The first word is the name of the command, and the remaining words are arguments to the command.

```
$foo
```

The dollar sign ($) substitutes the value of a variable. In this example, the variable name is foo.

```
[clock seconds]
```

Square brackets execute a nested command. For example, if you want to pass the result of one command as the argument to another, you use this syntax. In this example, the nested command is clock seconds, which gives the current time in seconds.

```
"some stuff"
```

Double quotation marks group words as a single argument to a command. Dollar signs and square brackets are interpreted inside double quotation marks.

```
{some stuff}
```

Curly brackets also group words into a single argument. In this case, however, elements within the brackets are not interpreted.

# Appendix D: Tcl/Tk Application

This is the Tcl/Tk implementation of the application. First the main application which is implemented in one single file, *cbgui.tcl*, then the module which also is implemented in one single file *li_gui.tcl*.

*cbgui.tcl*

```
#============================================================
# Name            : cbgui.tcl
#
# Component       :
#
# ------------------C o p y r i g h t-----------------------
#
#   Copyright (C) Telefonaktiebolaget LM Ericsson 2000.
#   The copyright to the computer program herein is the
#   property of Telefonaktiebolaget LM Ericsson Sweden.
#   The program may be used and/or copied only with the
#   written permission from Telefonaktiebolaget LM
#   Ericsson or in accordance with the terms and conditions
#   stipulated in the agreement/contract under which the
#   program has been supplied.
#
# ----------------------C r e a t e d-----------------------
#
# Created: March  2, 2000
# Creator: Peter Svensson
#
# ------------------D o c u m e n t a t i o n---------------
#   <add> files
#   <name>cbgui.tcl
#   ADD FILE DESCRIPTION HERE!!!
#   <end>
#
#============================================================
#

################################################################
#
# IMPORTS
#
################################################################

package require BWidget 1.2
package require MPH 2.0

################################################################
#
# GLOBAL VARIABLES
#
################################################################

variable menuPath

################################################################
#
# MPH PROCEDURES
#
################################################################

#-------------------------------------------------------------
# Procedure : SendMessage
# Abstract  : This procedure is used by the GUI instances to
#           : send text messages to 'its' SEA entity.
# Parameters: channel, the channel given to the instance
#           : message, the text message.
# Returns   : ---
#-------------------------------------------------------------
proc SendMessage {channel message} {
  global conId

  MPH::SendMessage $conId $channel $message
}

#-------------------------------------------------------------
# Procedure : SendBinaryMessage
# Abstract  : This procedure is used by the GUI instance to
```

```
#           : send binary messages to 'its' SEA entity.
# Parameters: channel, the channel given to the instance
#           : message, the binary message.
# Returns   : ---
#-------------------------------------------------------------
proc SendBinaryMessage {channel length message} {
  global conId

  MPH::SendBinaryMessage $conId $channel $length $message
}

#-------------------------------------------------------------
# Procedure : concloseproc
# Abstract  : This is a callback procedure required by MPH,
#           : it is called if the MPH connection is closed
# Parameters: ---
# Returns   : ---
#-------------------------------------------------------------
proc concloseproc {} {
  puts "MPH connection closed"
}

#############################################################
#
# GUI PROCEDURES
#
#############################################################

#-------------------------------------------------------------
# Procedure : draw
# Abstract  : This procedure creates and places the widgets,
#           : menubar and notebook, on the main window.
# Parameters: ---
# Returns   : ---
#-------------------------------------------------------------
proc draw {} {
  global menuPath

  frame .mbar -relief raised -bd 2
  pack .mbar -side top -fill x

  set menuPath .mbar.mbAdded.menu
  menubutton .mbar.mbAdded -text Tools -menu $menuPath
  menu $menuPath -tearoff 0
  pack .mbar.mbAdded -side left

  NoteBook .nb
  pack .nb -side left -expand 1 -fill both
}

#-------------------------------------------------------------
# Procedure : addToMenu
# Abstract  : Adds a new command to the given cascade menu
#           : in the menu pointed out by the global variable
#           : menuPath. If the cascade dont exist it is created
# Parameters: cascade, name of the cascade menu
#           : name, name of the new command
#           : command, code to be executed when the command is
#                      selected from the menu
# Returns   : ---
#-------------------------------------------------------------
proc addToMenu {cascade name command} {
  global menuPath

  if {![catch { menu ${menuPath}.cascade_$cascade -tearoff 0 } result]} {
    # If cascade 'mod' does not exist we create it.
    $menuPath add cascade -label $cascade -menu ${menuPath}.cascade_$cascade
  }

  ${menuPath}.cascade_$cascade add command -label $name -command $command
}

#-------------------------------------------------------------
# Procedure : killTab
# Abstract  : Deletes a tab from the notebook
# Parameters: ---
# Returns   : ---
#-------------------------------------------------------------
proc killTab {inst} {
  .nb delete $inst
}

#-------------------------------------------------------------
# Procedure : tearOff
# Abstract  : This procedure makes a new toplevel window,
#           : deletes the specified instance and adds it to
#           : the new toplevel window
```

```
# Parameters: inst, name of the instance, tab, to tear off
#           : channel, the MPH channel given to the module
#           : mod, type of instance
# Returns   : --
#-------------------------------------------------------------
proc tearOff {inst channel mod} {
  toplevel .tearoff$inst
  killTab $inst
  set tearOffClient [initTearOffPage .tearoff$inst $inst $channel $mod]
  ${inst}::init $tearOffClient $channel
}

#-------------------------------------------------------------
# Procedure : tearOn
# Abstract  : This procedure destroys a given toplevel window
#           : and creates a new tab in the notebook and initiates
#           : the module there.
# Parameters: inst, the name of the instance
#           : channel, the channel given to the GUI instance
#           : mod, the type of instance.
# Returns   : ---
#-------------------------------------------------------------
proc tearOn {inst channel mod} {
  destroy .tearoff$inst
  set clientPath [createTab $mod $inst $channel]
  .nb raise $inst
  ${inst}::init $clientPath $channel
}

#-------------------------------------------------------------
# Procedure : initTearOffPage
# Abstract  : This procedure initiates the new toplevel window
#           : and creates a frame for the module to draw its widgets in.
# Parameters: page,
#           : inst, the name of the instance
#           : channel, the channel given to the GUI instance
#           : mod, the type of instance.
# Returns   : The path to the drawing area for the module
#-------------------------------------------------------------
proc initTearOffPage {page inst channel mod} {
  frame $page.info
  frame $page.client -relief sunken -bd 2
  pack $page.info -side top -fill x
  pack $page.client -side top -expand 1 -fill both

  label $page.info.label -text "Inst: $inst"
  button $page.info.tearOff -text TearOn -command "tearOn $inst $channel $mod"
  pack $page.info.label -side left
  pack $page.info.tearOff -side right

  return $page.client
}

#-------------------------------------------------------------
# Procedure : initPage
# Abstract  : This procedure is called immediately after a new
#           : tab has been created. It creates widgets for some
#           : standard functionality and the frame that is by
#           : sent to the GUI instance.
# Parameters: mod, the type of instance.
#           : page, the tab page to add the frame to
#           : inst, the name of the instance
#           : channel, the channel given to the GUI instance
# Returns   : The frame that the GUI instance will use
#-------------------------------------------------------------
proc initPage {mod page inst channel} {
  frame $page.info
  frame $page.client -relief sunken -bd 2
  pack $page.info -side top -fill x
  pack $page.client -side top -expand 1 -fill both

  label $page.info.label -text "Inst: $inst"
  button $page.info.tearOff -text TearOff -command "tearOff $inst $channel $mod"
  button $page.info.remove -text Remove -command "destroyEntry $mod $inst $channel"
  pack $page.info.label -side left
  pack $page.info.tearOff $page.info.remove -side right
  return $page.client
}

#-------------------------------------------------------------
# Procedure : createTab
# Abstract  : This procedure is called immediately after a
#           : has been selected from the menu. It creates a
#           : new tab and call initPage to initiate it
# Parameters: mod, the type of instance.
#           : inst, the name of the instance
#           : channel, the channel given to the GUI instance
```

```
# Returns   : The frame that the GUI instance will use
#-------------------------------------------------------------
proc createTab {mod inst channel} {
  global menuPath

  $menuPath.cascade_$mod entryconfigure $inst -state disabled

  set page [.nb insert end $inst -text "${inst}" ]
  set clientPath [initPage $mod $page $inst $channel]
  .nb raise $inst
  return $clientPath
}

############################################################
#
# PROCEDURES
#
############################################################

#-------------------------------------------------------------
# Procedure : newEntry
# Abstract  : Called if the user selects a entity from the menu
#           : It loads the source code for the given instance
# Parameters: mod, the type of instance.
#           : inst, the name of the instance
#           : tclappsource, file containing the new source code
#           : to load
# Returns   : ---
#-------------------------------------------------------------
proc newEntry {mod inst tclappsource} {
  global conId

  namespace eval $inst {}

  proc ${inst}::loadtclappsource {inst} {
    source $inst
  }

  ${inst}::loadtclappsource $tclappsource
  set iid [${inst}::getIID]
  set channel [MPH::OpenChannel $conId $iid $inst \
                  ${inst}::messagecallback ${inst}::closecallback]
  set userpath [createTab $mod $inst $channel]
  ${inst}::init $userpath $channel
}

#-------------------------------------------------------------
# Procedure : destroyEntry
# Abstract  : Removes a GUI instance by disconnecting its MPH
#           : channel, enable it in the menu and remove its tab
# Parameters: mod, the type of instance.
#           : inst, the name of the instance
#           : channel, the channel given to the GUI instance
# Returns   : ---
#-------------------------------------------------------------
proc destroyEntry {mod inst channel} {
  global conId menuPath

  namespace delete $inst

  MPH::CloseChannel $conId $channel
  $menuPath.cascade_$mod entryconfigure $inst -state normal
  killTab $inst
}

#-------------------------------------------------------------
# Procedure : addEntities
# Abstract  : Checks available entities in SEA against the
#           : config file 'cbgui.cfg' and adds those entities
#           : that have a GUI to the menu
# Parameters: ---
# Returns   : ---
#-------------------------------------------------------------
proc addEntities {} {
  global conId

  set entities [MPH::SearchByName $conId ".*"]

  set file [open cbgui.cfg r]
  set supported [list]
  while { [gets $file line] >= 0} {
    regexp {(^[^:]+): *(.*)} $line junk mod tclfile
    foreach i $entities {
      if {[regexp ${mod}\[0-9\]$ $i match] == 1} {
        addToMenu $mod $match "newEntry $mod $match $tclfile"
      }
    }
```

```
  }
  close $file

  puts [lsort -increasing $supported]
}

#################################################################
#
# MAIN
#
#################################################################
wm geometry . 640x480

set conId closed

set seaHost [lindex $argv 0]
set seaPort [lindex $argv 1]

set conId [MPH::OpenConnection $seaHost $seaPort concloseproc]
puts "Using MPH connection on socket: $conId"

draw
addEntities

tkwait window .

if {$conId != "closed"} {
  MPH::CloseConnection $conId
}
```

## *li_gui.tcl*

```
#===============================================================
# Name            : li_gui.tcl
#
# Component       :
#
# ------------------C o p y r i g h t-------------------------
#
#   Copyright (C) Telefonaktiebolaget LM Ericsson 2000.
#   The copyright to the computer program herein is the
#   property of Telefonaktiebolaget LM Ericsson Sweden.
#   The program may be used and/or copied only with the
#   written permission from Telefonaktiebolaget LM
#   Ericsson or in accordance with the terms and conditions
#   stipulated in the agreement/contract under which the
#   program has been supplied.
#
# ----------------------C r e a t e d-----------------------
#
# Created: March  3, 2000
# Creator: Peter Svensson
#
# ------------------D o c u m e n t a t i o n----------------
#   <add> files
#   <name>li_gui.tcl
#   ADD FILE DESCRIPTION HERE!!!
#   <end>
#
#===============================================================
#

#################################################################
#
# IMPORTS
#
#################################################################


#################################################################
#
# GLOBAL VARIABLES
#
#################################################################


#################################################################
#
# NAMESPACES
#
```

```
#############################################################


#############################################################
#
# PROCEDURES CALLED FROM THE MAIN CONTROL APPLICATION
#
#############################################################


#-------------------------------------------------------------
# Procedure : getIID
# Abstract  : Returns the IID number of the module
# Parameters: None
# Returns   : Returns the IID of the module
#-------------------------------------------------------------
proc getIID {} {
  return 75bfa730-2e15-11d3-81b3-08002093ddf7
}


#-------------------------------------------------------------
# Procedure : messagecallback
# Abstract  : This procedure is called when a message has
#           : been received
# Parameters: length
#           : message
# Returns   : -
#-------------------------------------------------------------
proc messagecallback {length message} {
  set l [expr $length - 1]
  set l [expr $l * 8]
  binary scan $message "H2H*" cmd arg
  setInfotext "callBack: $cmd $arg"
}


#-------------------------------------------------------------
# Procedure : closecallback
# Abstract  : This procedure is called if the channel is
#           : closed remotely by the SEA component.
# Parameters: -
# Returns   : -
#-------------------------------------------------------------
proc closecallback {} {

}


#-------------------------------------------------------------
# Procedure : init
# Abstract  : This procedure is called when the module is
#           : instantiated.
# Parameters: _path,    The modules virtual root to draw its
#           :           widgets in.
#           : _channel, the MPH channel id, connected to the
#           :
# Returns   : -
#-------------------------------------------------------------
proc init {_path _channel} {
  variable [namespace current]::channel
  variable [namespace current]::path

  set channel $_channel
  set path $_path

  frame ${path}.numbers -relief sunken -bd 2

  button ${path}.numbers.offhock -text "Off Hock" -command [namespace current]::offHook
  button ${path}.numbers.onhock -text "On Hock" -command [namespace current]::onHook
  pack ${path}.numbers.offhock ${path}.numbers.onhock -padx 3 -pady 3 -fill x

  frame ${path}.numbers.one
  foreach i {1 2 3} {
    button ${path}.numbers.one.$i -text $i -command "[namespace current]::dial $i"
  }
  pack ${path}.numbers.one.1 ${path}.numbers.one.2 ${path}.numbers.one.3 -side left

  frame ${path}.numbers.two
  foreach i {4 5 6} {
    button ${path}.numbers.two.$i -text $i -command "[namespace current]::dial $i"
  }
  pack ${path}.numbers.two.4 ${path}.numbers.two.5 ${path}.numbers.two.6 -side left

  frame ${path}.numbers.three
  foreach i {7 8 9} {
    button ${path}.numbers.three.$i -text $i -command "[namespace current]::dial $i"
```

```
  }
  pack ${path}.numbers.three.7 ${path}.numbers.three.8 ${path}.numbers.three.9 -side left

  button ${path}.numbers.zero -text "0" -command "[namespace current]::dial 0"
  pack ${path}.numbers.one ${path}.numbers.two ${path}.numbers.three ${path}.numbers.zero

  frame ${path}.info
  text ${path}.info.info -relief sunken -bd 2 -yscrollcommand "${path}.info.scroll set" -height 15 -width 16
-state disabled
  scrollbar ${path}.info.scroll -command "${path}.info.info yview"
  pack ${path}.info.scroll -side right -fill y
  pack ${path}.info.info -fill both -expand yes

  pack ${path}.numbers -padx 3 -pady 3 -side left
  pack ${path}.info -padx 3 -pady 3 -side left -fill both -expand yes
}

############################################################
#
# COMPONENT SPECIFIC PROCEDURES
#
############################################################
variable channel
variable path

#-----------------------------------------------------------
# Procedure : offHook
# Abstract  : Sends a offHook message to its SEA component.
# Parameters: -
# Returns   : -
#-----------------------------------------------------------
proc offHook {} {
  variable [namespace current]::channel

  set OFF_HOOK [binary format "H2" 00]
  SendBinaryMessage $channel 1 $OFF_HOOK
  setInfotext "Phone off the hook"
}

#-----------------------------------------------------------
# Procedure : onHook
# Abstract  : Sends a onHook message to its SEA component.
# Parameters: -
# Returns   : -
#-----------------------------------------------------------
proc onHook {} {
  variable [namespace current]::channel

  set ON_HOOK [binary format "H2" 01]
  SendBinaryMessage $channel 1 $ON_HOOK
  setInfotext "Phone on the hook"
}


#-----------------------------------------------------------
# Procedure : dial
# Abstract  : Sends a selected digit to its SEA component.
# Parameters: digit, digit to send.
# Returns   : -
#-----------------------------------------------------------
proc dial {digit} {
  variable [namespace current]::channel

  switch $digit {
    0 {set bin [binary format "H4" 0300]}
    1 {set bin [binary format "H4" 0301]}
    2 {set bin [binary format "H4" 0302]}
    3 {set bin [binary format "H4" 0303]}
    4 {set bin [binary format "H4" 0304]}
    5 {set bin [binary format "H4" 0305]}
    6 {set bin [binary format "H4" 0306]}
    7 {set bin [binary format "H4" 0307]}
    8 {set bin [binary format "H4" 0308]}
    9 {set bin [binary format "H4" 0309]}
  }

  SendBinaryMessage $channel 2 $bin
  [namespace current]::setInfotext "Dial $digit"
}


#-----------------------------------------------------------
# Procedure : setInfotext
# Abstract  : Prints text in the textarea
# Parameters: text, text to print.
# Returns   : -
#-----------------------------------------------------------
```

```
proc setInfotext {text} {
  variable [namespace current]::path

  ${path}.info.info configure -state normal
  ${path}.info.info insert end "${text}\n"
  ${path}.info.info see end
  ${path}.info.info configure -state disabled
}
```

# Appendix E: Java Application

This is the Java applet implementation of the application. First the main application which is implemented as three classes, CbguiApplet, ComponentList and Tabs, then the module which is implemented in one single class, LI.

The main Java application.

### *Class CbguiApplet*

```
//## begin module.cm preserve=no
//   %X% %Q% %Z% %W%
//## end module.cm

/* IMPORTS */
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;
import MPHclient.*;
/* INTERFACE DEFINITIONS */

/**********************************************************
* Class      : CbguiApplet
* Extends    : JApplet
* Implements : ---
* Abstract   : The main Cbgui Applet is implemented as an applet.
**********************************************************/
public class CbguiApplet extends JApplet {
  ComponentList compList = new ComponentList(this);
  Tabs tabs = new Tabs(this);
  MPHclient mph;

  /**********************************************************
  * Method    : addComponents
  * Abstract  : This method is meant to get the names of the loaded
  *             components in the SEA core and check if these
  *             components have an associated module class. If they
  *             have a module class they are added to the component
  *             list. But since the Java MPH library does not have
  *             any functionality that makes it possible to get
  *             the loaded components from SEA this method just
  *             adds a few known component names to the list.
  * Parameters: ---
  * Returns   : ---
  **********************************************************/
  public void addComponents() {
    compList.newEntry("LI", "LI-0");
    compList.newEntry("LI", "LI-1");
    compList.newEntry("LI", "LI-2");
    compList.newEntry("LI", "LI-3");
    compList.newEntry("LI", "LI-4");
    compList.newEntry("LI", "LI-5");
    compList.newEntry("LI", "LI-6");
    compList.newEntry("LI", "LI-7");
    compList.newEntry("AT", "AT-0");
    compList.newEntry("AT", "AT-1");
    compList.newEntry("AT", "AT-2");
    compList.newEntry("AT", "AT-3");
    compList.newEntry("AT", "AT-4");
    compList.newEntry("AT", "AT-5");
    compList.newEntry("AT", "AT-6");
    compList.newEntry("AT", "AT-7");
  }

  /**********************************************************
  * Method    : init
  * Abstract  : This method is called when the applet is created.
  *             It gets the host and port for the running SEA and
  *             sets up the MPH connection to it.
  * Parameters: ---
  * Returns   : ---
  **********************************************************/
  public void init() {
    String host = getParameter("host");
```

```
    int port = (Integer.decode(getParameter("port"))).intValue();

    System.out.println("Host: " + host);
    System.out.println("Port: " + port);

    try{
      mph = new MPHclient(host, port);
    }
    catch(MPHException e){
      System.out.println("Failed to establish connection to "
                         + host + ":" + port);
      System.out.println("Error: " + e.getMessage());
      return;
    }

    System.out.println("MPH client created to Host: "
                       + host + "on Port: " + port);
    addComponents();
  }

  /************************************************************
  * Method    : itemSelectCallback
  * Abstract  : This is a callback method that get called by the
  *             componentList object when a component has been
  *             selected from the component list. This method
  *             loads and instantiates a new object of type 'mod',
  *             sets up a new MPHconnection to component 'inst',
  *             calls the addComponent in class tabs to get a new
  *             JPanel in a new tab in the JTabbedPane and finally
  *             calls the method init in the newly created object
  *             with the MPHconnection and the JPanel as parameters.
  * Parameters: String mod, name of the class to load.
  *             String inst, name of the SEA component to connect to.
  * Returns   : ---
  ************************************************************/
  public void itemSelectCallback(String mod, String inst) {
    CompInterface module;
    MPHconnection mphConn;

    try {
      java.lang.Class t = Class.forName(mod);
      module = (CompInterface)t.newInstance();
    }
    catch (Throwable e) {
      System.out.println("Could not load or instanciate " + mod + ".class");
      System.out.println("Error: " + e.getMessage());
      return;
    }
    System.out.println("Loaded " + mod + ".class");

    String iid = module.compGetIID();
    try{
      mphConn = mph.connect(inst, iid,  (MPHclientListener)module);
    }
    catch(MPHException e){
      System.out.println("Failed to connect to " + inst);
      System.out.println("Error: " + e.getMessage());
      return;
    }

    JPanel panel = tabs.addComponent(inst);
    module.compInit(mphConn, panel);
  }

  /************************************************************
  * Method    : destroy
  * Abstract  : This method is called when the applet is destroyed.
  *             It close the MPHconnection.
  * Parameters: ---
  * Returns   : ---
  ************************************************************/
  public void destroy() {
    try{
      mph.closeConnection();
    }
    catch(MPHException e){
      System.out.println("Failed to close connection");
      System.out.println("Error: " + e.getMessage());
    }
    System.out.println("MPH socket closed");
  }
}
```

## *Class ComponentList*

```
/* IMPORTS */
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

/************************************************************
* Class      : ComponentList
* Extends    : ---
* Implements : ActionListener
* Abstract   : This class handles the representation of the
*              names of the loaded components in the SEA
*              core. The list of loaded components is implemented
*              as a menu.
************************************************************/
public class ComponentList implements ActionListener {
  JMenuBar menuBar;
  JMenu menu;
  CbguiApplet owner;

  /************************************************************
  * Method    : ComponentList (Constructor)
  * Abstract  : Creates the menubar and adds a menu to it. The
  *             menubar is then added to the main applet window.
  * Parameters: CbguiApplet cb, the object that instantiate
  *             the class
  * Returns   : ---
  ************************************************************/
  public ComponentList(CbguiApplet cb){
    owner = cb;
    menuBar = new JMenuBar();
    owner.setJMenuBar(menuBar);

    menu = new JMenu("Tools");
    menu.setMnemonic(KeyEvent.VK_C);
    menu.getAccessibleContext().setAccessibleDescription("The only menu");
    menuBar.add(menu);
  }

  /************************************************************
  * Method    : newEntry
  * Abstract  : Adds new entries to the menu. The new entry 'inst'
  *             is placed in the cascade menu 'mod'. If cascade mod
  *             does not exist it is created. Since this class implements
  *             actionlistener the action for every new entry is set to
  *             this.
  * Parameters: String mod, name of the cascade menu.
  *             String inst, name of the new entry.
  * Returns   : ---
  ************************************************************/
  public void newEntry(String mod, String inst) {
    JMenu subMenu = null;
    JMenuItem menuItem;
    int count;

    count = menu.getItemCount();

    for(int i=0; i < count; i++){
      if(menu.getItem(i).getText().equals(mod)){
        subMenu = (JMenu)menu.getItem(i);
      }
    }
    if(subMenu == null) {
      subMenu = new JMenu(mod);
      menu.add(subMenu);
    }

    menuItem = new JMenuItem(inst);
    menuItem.addActionListener(this);

    subMenu.add(menuItem);
  }


  /************************************************************
  * Method    : actionPerformed
  * Abstract  : Declared in the ActionListener interface. Define
  *             the action for the menu entries. The action performed
  *             is calling the owner class callback method
  *             itemSelectCallback with the type and name of the
  *             selected menu entry.
  * Parameters: ActionEvent e.
  * Returns   : ---
  ************************************************************/
  public void actionPerformed(ActionEvent e) {
```

```
    JMenuItem source = (JMenuItem)(e.getSource());

    owner.itemSelectCallback(source.getText().substring(0,2), source.getText());
    source.setEnabled(false);
  }
}
```

## *Class Tabs*

```
/* IMPORTS */
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

/* INTERFACE DEFINITIONS */

/*********************************************************
* Class      : Tabs
* Extends    : JTabbedPane
* Implements : ---
* Abstract   : This class handles the module classes drawing
*              areas. The drawing areas are JPanels placed in
*              a JTabbedPane
*********************************************************/
public class Tabs extends JTabbedPane {

  /*********************************************************
  * Method    : Tab (Constructor)
  * Abstract  : Adds the JTabbedPane to the main class window
  * Parameters: JApplet parent, the object that instantiate
  *             the class
  * Returns   : ---
  *********************************************************/
  public Tabs(JApplet parent){
    parent.getContentPane().add(this);
  }

  /*********************************************************
  * Method    : addComponent
  * Abstract  : Creates a new JPanel, adds this to a new tab
  *             in the JTabbedPane named 'inst' and returns
  *             the JPanel.
  * Parameters: String inst, the name for the new tab
  * Returns   : The created JPanel
  *********************************************************/
  public JPanel addComponent(String inst) {
    JPanel panel = new JPanel();

    addTab(inst, panel);

    return panel;
  }
}
```

## *Class LI*

```
/* IMPORTS */
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import MPHclient.*;
/* INTERFACE DEFINITIONS */

/*********************************************************
* Class      : LI
* Extends    : ---
* Implements : CompInterface, MPHclientListener, ActionListener
* Abstract   : This class is to be used as a dynamically loaded
*              class by the CbguiApplet. It is a simple test
*              implementation to communicate with a LIC component
*              in SEA. The GUI consist of a simple keypad and a
*              textfield
*********************************************************/
public class LI
  implements CompInterface, MPHclientListener, ActionListener {
  MPHconnection mphConn;
  JTextArea text;

  /*********************************************************
  * Method    : compGetIID
```

```
* Abstract  : Declared in the CompInterface interface. It
*             returns the identifier ID for the LIC component.
* Parameters: ---
* Returns   : The IID for the LIC component
*************************************************************/
public String compGetIID() {
  return "75bfa730-2e15-11d3-81b3-08002093ddf7";
}

/*************************************************************
* Method    : compInit
* Abstract  : Declared in the CompInterface interface. This method
*             is called after an instance of this class has been
*             created, It is used for setting up the GUI used to
*             control the LIC component.
* Parameters: MPHconnection mph, an instance of the MPHconnection
*             used to communicate with the LIC.
*           : JPanel p, this is the JPanel that this class must
*             use for all user interaction
* Returns   : ---
*************************************************************/
public void compInit(MPHconnection mph, JPanel p){
  JButton jb;

  mphConn = mph;

  JPanel numPad = new JPanel();
  numPad.setLayout(new GridLayout(4, 3));
  numPad.add(jb = new JButton("1")); jb.addActionListener(this);
  numPad.add(jb = new JButton("2")); jb.addActionListener(this);
  numPad.add(jb = new JButton("3")); jb.addActionListener(this);
  numPad.add(jb = new JButton("4")); jb.addActionListener(this);
  numPad.add(jb = new JButton("5")); jb.addActionListener(this);
  numPad.add(jb = new JButton("6")); jb.addActionListener(this);
  numPad.add(jb = new JButton("7")); jb.addActionListener(this);
  numPad.add(jb = new JButton("8")); jb.addActionListener(this);
  numPad.add(jb = new JButton("9"));  jb.addActionListener(this);
  numPad.add(jb = new JButton("Off"));jb.addActionListener(this);
  numPad.add(jb = new JButton("0"));  jb.addActionListener(this);
  numPad.add(jb = new JButton("On")); jb.addActionListener(this);

  text = new JTextArea();
  text.setEditable(false);

  JScrollPane editorScrollPane = new JScrollPane(text);
  editorScrollPane.setVerticalScrollBarPolicy(
                          JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
  editorScrollPane.setPreferredSize(new Dimension(250, 145));

  p.add(numPad);
  p.add(editorScrollPane);
}

/*************************************************************
* Method    : receiveMessage
* Abstract  : Declared in the MPHclientListener interface. This
*             is a callback method used by the MPH to pass messages
*             to this instance. The message comes as a byte array
*             that is checked and printed in the textfileld.
* Parameters: byte[] data, the message.
*           : int length, number of bytes in the message
* Returns   : ---
*************************************************************/
public void receiveMessage(byte[] data, int length){
  String inData;

  switch((int)data[0]) {
  case 0: inData = "RING SIGNAL\n"; break;
  case 1: inData = "STOP RING SIGNAL\n"; break;
  case 2: inData = "TONE\n"; break;
  case 3: inData = "STOP TONE\n"; break;
  case 4: inData = "CONNECT INFO\n"; break;
  case 5: inData = "IDLE PHONE\n"; break;
  case 6: inData = "SPEECH DATA LIC\n"; break;
  case 7: inData = "VIRTUAL SECOND\n"; break;
  case 8: inData = "SPEECH AS TEXT\n"; break;
  case 9: inData = "FSK MESSAGE\n"; break;
  default: inData = "Unkown message: "
             + Integer.toString((int)data[0]) +"\n";
  }

  text.append(inData);
}

/*************************************************************
* Method    : connectionClosed
* Abstract  : Declared in the MPHclientListener interface.
```

```
  *             Callback method that is called if the MPH connection
  *             is closed.
  * Parameters: ---
  * Returns   : ---
  ************************************************************/
  public void connectionClosed(){
  }

  /************************************************************
  * Method    : channelClosed
  * Abstract  : Declared in the MPHclientListener interface.
  *             Callback method that is called if the MPH channel
  *             is closed.
  * Parameters: ---
  * Returns   : ---
  ************************************************************/
  public void channelClosed(){
  }

  /************************************************************
  * Method    : error
  * Abstract  : Declared in the MPHclientListener interface.
  *             Callback method that is called if there is an
  *             error in the communication with MPH.
  * Parameters: String errMsg, error message.
  * Returns   : ---
  ************************************************************/
  public void error(String errMsg){
  }

  /************************************************************
  * Method    : actionPerformed
  * Abstract  : Declared in the ActionListener interface. Define
  *             the action for the buttons in the keypad.
  * Parameters: ActionEvent e.
  * Returns   : ---
  ************************************************************/
  public void actionPerformed(ActionEvent e) {
    JButton source = (JButton)(e.getSource());
    String button = source.getText();
    byte action[] = new byte[3];
    int length = 0;
    String info;

    if(button.equals("Off")){
      action[0] = 0;
      length = 1;
      info = "Phone is off hook\n";
    } else if(button.equals("On")){
      action[0] = 1;
      length = 1;
      info = "Phone is on hook\n";
    } else switch((Integer.valueOf(button)).intValue()){
    case 0: action[0] = 3; action[1] = 0; length = 2; info = "Dial 0\n"; break;
    case 1: action[0] = 3; action[1] = 1; length = 2; info = "Dial 1\n"; break;
    case 2: action[0] = 3; action[1] = 2; length = 2; info = "Dial 2\n"; break;
    case 3: action[0] = 3; action[1] = 3; length = 2; info = "Dial 3\n"; break;
    case 4: action[0] = 3; action[1] = 4; length = 2; info = "Dial 4\n"; break;
    case 5: action[0] = 3; action[1] = 5; length = 2; info = "Dial 5\n"; break;
    case 6: action[0] = 3; action[1] = 6; length = 2; info = "Dial 6\n"; break;
    case 7: action[0] = 3; action[1] = 7; length = 2; info = "Dial 7\n"; break;
    case 8: action[0] = 3; action[1] = 8; length = 2; info = "Dial 8\n"; break;
    case 9: action[0] = 3; action[1] = 9; length = 2; info = "Dial 9\n"; break;
    default: info = "error";
    }

    try{
      mphConn.send(action, length);
    }
    catch(MPHException c){
      System.out.println("Faild to send message" + button);
      System.out.println("Error: " + c.getMessage());
      return;
    }
    text.append(info);
  }
}
```

## *interface CompInterface*

```
/* IMPORTS */
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import MPHclient.*;
/* INTERFACE DEFINITIONS */

/* TYPE DEFINITIONS */

/**********************************************************
* Interface  : CompInterface
* Abstract   : This interface has to be implemented by a class
*              that is to be dynamically loaded by the CbguiApplet
**********************************************************/
public interface CompInterface {

  /**********************************************************
  * Interface : compInit
  * Abstract  : This method is called after the class has been
  *             initiated.
  * Parameters: MPHconnection mph, an instance of the MPHconnection
  *             used to communicate with a given SEA component.
  *           : JPanel p, this is the JPanel that this class must
  *             use for all user interaction
  * Returns   : ---
  **********************************************************/
  public void compInit(MPHconnection mph, JPanel p);

  /**********************************************************
  * Method    : compGetIID
  * Abstract  : This method must return identifier ID for the
  *             given component.
  * Parameters: ---
  * Returns   : The IID for the given component.
  **********************************************************/
  public String compGetIID();
}
```