

MSc Distributed Computing System Engineering  
Department of Electronic & Computer  
Engineering

## **Brunel University**

Designing and building a graphical  
"design-toolkit" for the  
"Grasshopper" mobile agent system.

Sascha Heinisch

**September / 2000**

A Dissertation submitted in partial fulfilment of the  
requirements for the degree of Master of Science

# Acknowledgements

I would like to thank to my reviewers who took the time to read this dissertation through and to the many people who were willing to discuss the various problems which had been raised during this dissertation.

This dissertation has required much time to be written. Therefore I express my thanks to the company DaimlerChrysler and to my boss Mr. Goll for allowing me to spend so much time in this dissertation. Also many thanks to my supervisor Dr. Manissa Wilson at the Brunel University for her help and advice. She managed the difficult task to accompany this dissertation via e-mail.

Leonberg, in September 2000

Sascha Heinisch

## Notation

New expressions as well as important new words are first written in "quotes". Words whose meanings are not directly explained are listed up in the glossary. Program code as well as class names are written in `courier new`.

# Abstract

In this work the development of an agent design–toolkit for the Grasshopper agent system is shown. This design–toolkit has an intuitive graphical user interface, that allows especially novices to build agents for their special need, without knowledge about programming. Also the user needs minimal knowledge about the principles of agent systems. All this knowledge has to be transferred into the design–toolkit, which has to supervise the design process and to inform the user as early as possible about errors he or she has done.

Because Grasshopper bases completely on Java the development of the design–toolkit is done by using Java together with the application programmers interface Swing and the graphical design toolkit Forte. To get a concept how to realise the graphical user interface the agent system as well as agent programming have to be analysed and split up into logical groups. To achieve a good intuitive appearance of the interface also some research for the basics of good interface design is done.

<b>1</b>	<b>SUMMARY</b>	<b>3</b>
<b>2</b>	<b>INTRODUCTION TO THE MASTER THESIS</b>	<b>4</b>
2.1	GENERAL	4
2.2	AIMS AND OBJECTIVES	5
<b>3</b>	<b>SWING</b>	<b>7</b>
3.1	WHAT IS SWING IN GENERAL	7
3.2	THE DELEGATE MODEL	7
3.3	PANES	9
3.4	A SMALL EXAMPLE	10
<b>4</b>	<b>FORTE FOR JAVA</b>	<b>12</b>
4.1	FORTE FOR JAVA IN GENERAL	12
4.2	EXAMPLE APPLICATION DESIGNED WITH FORTE	13
4.3	COMPARISON OF THE SOURCECODE FROM SECTION 2.4 AND 3.2	18
<b>5</b>	<b>THE GRASSHOPPER AGENT SYSTEM</b>	<b>19</b>
5.1	INTRODUCTION TO THE AGENT TECHNOLOGY	19
5.2	WHY USING THE GRASSHOPPER SYSTEM	21
5.3	THE GRASSHOPPER DISTRIBUTED AGENT ENVIRONMENT	22
<b>6</b>	<b>HOW TO PROGRAM AN AGENT</b>	<b>29</b>
6.1	BASICS	29
6.2	A SIMPLE MOVING AGENT	29
6.3	STORING DATA WITHIN AN AGENT	31
6.4	MIGRATION RESISTANT CONDITIONS OF AN AGENT	31
6.5	PROXY COMMUNICATION BETWEEN AGENTS	33
<b>7</b>	<b>THE IDENTIFIED FUNCTIONAL BLOCKS</b>	<b>39</b>
7.1	THE MAIN TEMPLATE	39
7.2	THE MOVE-BLOCK	40
7.3	THE AGENCY/PLACE INFO BLOCK	40
7.4	THE AGENT INFO BLOCK	41
7.5	COMMUNICATION BLOCKS	42
7.6	ADDITIONAL NON GRASSHOPPER SPECIFIC BLOCKS	43
<b>8</b>	<b>DESIGN OF THE GRAPHICAL USER INTERFACE</b>	<b>45</b>
8.1	ADVANTAGES AND RISKS OF GRAPHICAL USER INTERFACES	45
8.2	GENERAL DESIGN GUIDELINES	45
8.3	THE GUI FOR THE AGENT DESIGNER	48
8.4	FINDING A GRAPHICAL REPRESENTATION	50
8.5	THE REPEAT UNTIL CONSTRUCT	56
8.6	ANNOTATION TO THE DESIGN	57
<b>9</b>	<b>CONCLUSION AND FURTHER WORK</b>	<b>58</b>
9.1	CONCLUSION	58
9.2	FURTHER WORK	58
<b>10</b>	<b>ABBREVIATIONS</b>	<b>60</b>

**11GLOSSARY 61**

**12BIBLIOGRAPHY 62**

**13MANAGEMENT OF THE PROJECT 63**

13.1THE LEGEND OF THE TIME PLAN 63

13.2COMPARISON OF THE TIME PLAN WITH THE REAL USED TIME 63

# 1 SUMMARY

In this dissertation a graphical design toolkit for designing agents is developed. These agents are running under the Java based Grasshopper agent system. To accomplish this aim, the following techniques and tools are used:

- Using the Java Swing API for programming graphical user interfaces.
- Forte for Java, a graphical interface and program designer.
- The Grasshopper System.
- Agent programming with Grasshopper.
- General methods for designing graphical user interfaces.

The resulting design toolkit provides an intuitive user interface that allows non technical oriented users to design their own agent, without knowledge about agent or Java programming. This goal is achieved by representing the single code parts of an agent (further called "functional blocks" ) in an graphical way that allows the user to associate the building process with an easy puzzle or Lego. Due to restrictions of the Grasshopper system not every graphical part of the puzzle can be designed completely unambiguous.

## **2 INTRODUCTION TO THE MASTER THESIS**

### **2.1 GENERAL**

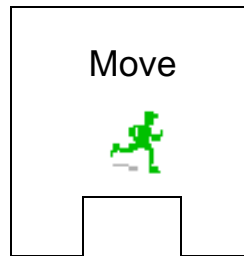
If anybody wants to get information from the internet he or she has only one possibility today to obtain it. He or she has to connect his PC with the internet and then to access it manually. Manually means that the user perhaps knows any WWW-Address of a site with the wanted content. This address has to be typed in, or perhaps the user may use a search engine like "Yahoo" that provides a keyword search of the content of huge areas of the internet. This way of obtaining information is very time consuming as well as expensive if a telephone line is used to connect the user computer with the internet. Very often a keyword search has some thousands of results that can not be reduced by adding new keywords to the search engine. The only thing the user can do is to select some search results by intuition.

In such a case the agent-technology will help. The user has to take one agent suitable for his or her special purposes and send it to the internet. At first the agent will do nearly the same in the internet as a human, it will try to get information of where the sites containing the desired information resides. The next step is to visit all these sites, more exactly to visit the servers where the sites are located, and to store the desired information. The third step is to transport the stored information back to the user. It is very likely that the agent will meet in the internet another agent on its way, that has to do nearly the same task. In that case these agents may be able to exchange their information about where to find good sites, which sites are not worth visiting and so on. Instead of search engines it would be more useful to provide information places for agents where they can obtain or exchange information about useful sites.

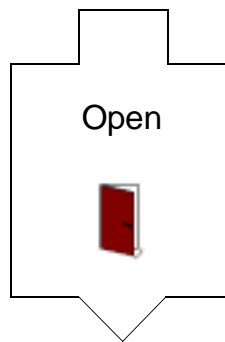
The main problem for the user is to get a suitable agent for his or her purpose. Today's agent systems agents need to be developed in Java, C, C++ or in C like scripting languages. This is very difficult for non technical oriented users and not acceptable. The solution for this problem is a graphically oriented design-toolkit that allows everybody to build their own special purpose agent without knowledge about programming nor the functionality of the used agent system.

## 2.2 AIMS AND OBJECTIVES

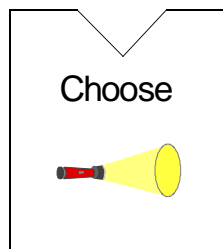
This master thesis explains how a design-toolkit can be built, as intuitive for the user as possible but with a minimal loss of the usable functionality provided by the agent system. The goal is that the user only combines graphic objects via drag and drop with the mouse and that he or she has not to type in "cryptographic" Java code. The single graphical objects should have a shape that avoids to combine the wrong objects with each other. So the user can recognise visually, which parts are possible to combine and which ones should be grouped in another order. For example an agent has three functions all represented by an graphical object. The first thing it may do is to migrate to its destination place in the network so we need a move block:



The second step for the agent is to open one of the perhaps various information data bases that may be available:

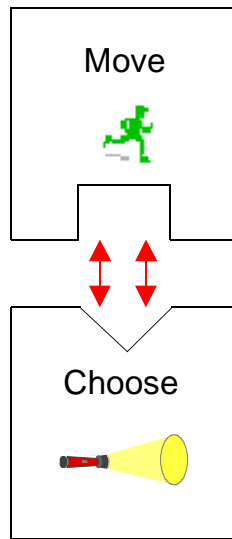


The third step is to choose the information from the database the agent has to carry back to the user:



If the user wants to combine "Move" with "Choose" it is for him or her easy to recognise that this is not possible.





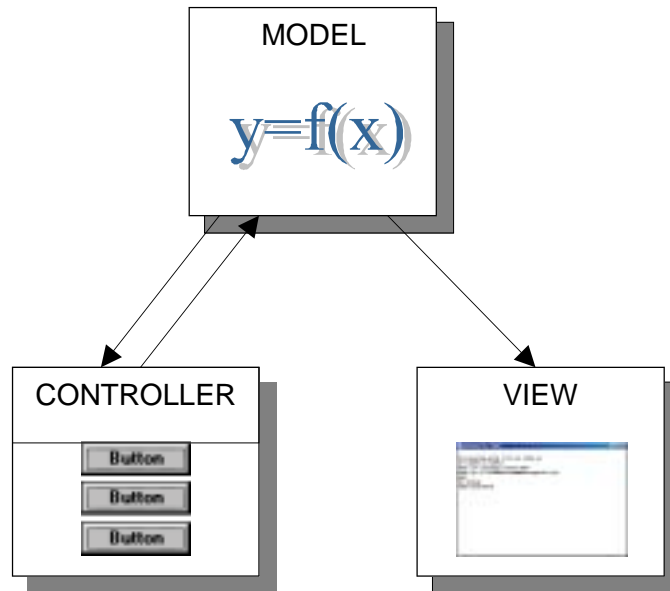
## **3 Swing**

### **3.1 WHAT IS SWING IN GENERAL**

Swing, Drag and Drop, the Accessibility APIs and Java 2D are the main parts of Java Foundation Classes (JFC). These classes in general implement nearly everything a programmer wants for 2D graphic and graphical user interface (GUI) programming. Swing especially is the part of the JFC that provides several "ready to use" GUI components like windows, buttons, scrollbars etc. One important feature of Swing is a pluggable look and feel that allows GUI-components to appear like any GUI of the today's usual operating systems. The word "Look" of the phrase "Look and Feel" means the appearance of the graphical user interface, the word "Feel" means the reaction of GUI-components like buttons on user activities. Swing is completely programmed in Java, based on the abstract windowing toolkit (AWT). All Swing components are called "lightweight components" because they are not dependent on the functionality provided by the operating system on which they are executed. In contrast AWT provides mainly "heavyweight components" that are based on the GUI-facilities provided by the operating system. The disadvantage of heavyweight components is, that AWT is only able to offer an subset of components that are equally supported by all usual OS platforms.

### **3.2 THE DELEGATE MODEL**

Swing is based on the "Delegate Model" that can be considered as a simplification of the "Model View Controller" (MVC) design pattern. The MVC pattern was first supported by the programming language Smalltalk. MVC is used to build interactive programmes that are able to get input for example by a keyboard, to do some calculation with the inputted data and to display the results on a suitable output device (Figure 3.1).



*Figure 3-1: the "Model View Controller" design pattern*

The advantage of MVC is that the three parts are doing their communication with well-known protocols, so it is possible to replace them individually. The benefit is a maximum of flexibility without reducing the latency or the speed of the system. That is possible because the MVC pattern does not dictate the transmission path for the protocol. The communication may be realised by function calls, inline Functions, Macros, TCP/IP or something else. The Java "Delegate Model" (Figure 3.2) is the result of observing what is needed in real applications. In nearly every application with a graphical user interface there is no separation between the "Controller" and the "View", only the model is an extra logical instance. This makes sense because the intention of Swing is to support programming user interfaces for common applications.

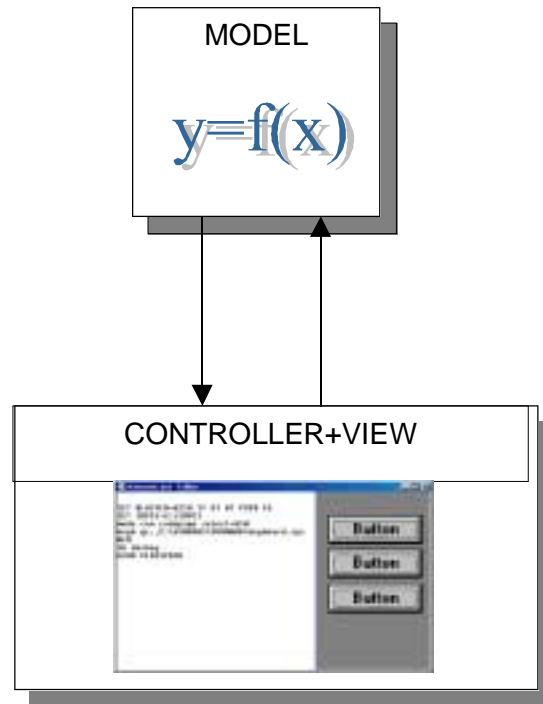
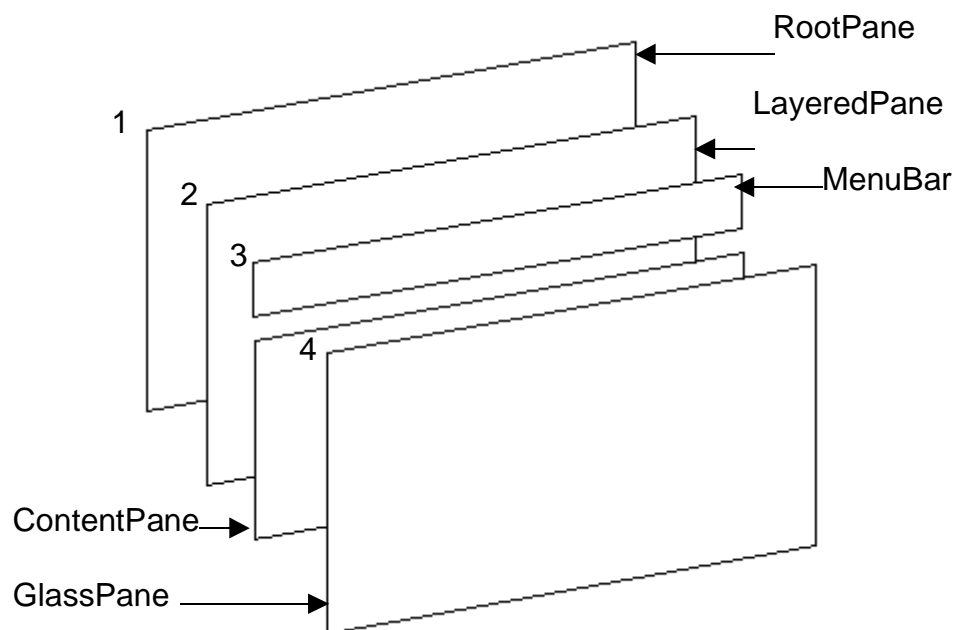


Figure 3-2: the Java "Delegate Model"

### 3.3 PANES

Swing holds the content of windows, like buttons or text areas, in panes. There exists a hierarchy of panes, each one with a special dedicated functionality. Every pane is realised with the Java collection classes. These collection classes containing Swing elements are called "container". Figure 3.3 shows this hierarchy.



*Figure 3–3: the Swing pane hierarchy*

Each different layer has its special purpose:

- **Rootpane**

The RootPane covers the LayeredPane as well as the GlassPane. It is the root of the container hierarchy.

- **LayeredPane**

The LayeredPane covers the ContentPane and optionally the MenuBar.

- **ContentPane**

The ContentPane is the container of most of the visible components excluding the MenuBar and the GlassPane.

- **MenuBar**

The MenuBar is an optional pane. It provides a space-saving area for user inputs.

- **GlassPane**

The GlassPane is a non-visible container in front of all panes. It is used to catch events or to draw over an area that already contains elements in the ContentPane.

The visibility of components in the different containers depends on their position in the hierarchy. Components in the GlassPane have the highest priority, so all other components that are located in containers below are overwritten. In Figure 3.3 the hierarchy is shown by the numbers 1 to 4.

### 3.4 A SMALL EXAMPLE

The following example, that can be found in nearly all Swing introductions, creates a window containing a button in the ContentPane. It should give an idea of how to use Swing.

```
import java.awt.*;
import javax.swing.*;
public class SwingButton
{
    public static void main (String[] args)
    {
        //create a new Window
        JFrame frame = new JFrame("Demo Application");

        //get the Contentpane of the window
        Container cont = frame.getContentPane();

        //create a button
        JButton button = new JButton("I am a Swing-button!");
```

```

//create a new layoutmanager
LayoutManager layout=new FlowLayout();

//set the layout for the contentpane
cont.setLayout(layout);

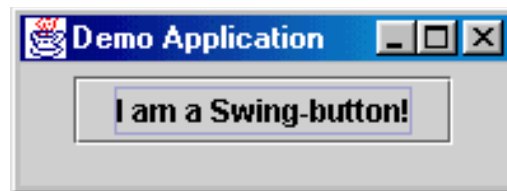
//put the button into the contentpane
cont.add(button);

//set size of the window
frame.setSize(200,75);

//show the window
frame.show();
}
}

```

The red coloured part of the program creates the objects for the window (JFrame), the ContentPane (Container) as well as the button object (JButton). The blue part creates the links between the classes and sets some values like the window size. The result of the program can be seen in Figure 3–4:



*Figure 3–4: The output of the SwingButton example*

Of course the only use of this program is to demonstrate Swing. It does not react on events like pressing the button. However there is a preinstalled reaction on pressing the "default" buttons in the frame of the window.

## 4 Forte for Java

### 4.1 FORTE FOR JAVA IN GENERAL

An integrated development environment (IDE) provides a more comfortable use of compilers, debugging and system tools compared to command line operations. It normally has a graphical user interface that encapsulates and hides all command line operations necessary to compile, debug, edit etc. The result is that the user has the impression that there only exists one big programme with many functions. This reduces the training time and enables a more fluent or faster working style. Forte is such an integrated development environment for Java. Beneath today's standard IDE–functionality it provides the following services:

- **Object Browser**

The Object Browser presents all sources in a logical streamlined view. It allows an intuitive working with classes by accessing their properties, compiling them etc.

- **Java–Editor**

The Editor is especially designed for editing Java sources. It provides syntax highlighting, code completion, line numbering etc. Also a printer support is available that allows printouts similar to the representation of the editor.

- **GridBag layout customizer**

The most complex Java layoutmanager is supported by a graphical user interface. It provides a dynamic visual representation of the components as well as adjusting them.

- **Update Center**

This feature allows an automated download of new modules for Forte via the internet.

- **Debugger**

A full graphical debugging support is available that approximately corresponds to the standard debugging tools for languages like C,C++ or Pascal.

- **Compiling and Linking**

All compiler and linker functions are supported. Code generation as

well as execution of the (bytecode-) executables from the GUI are supported.

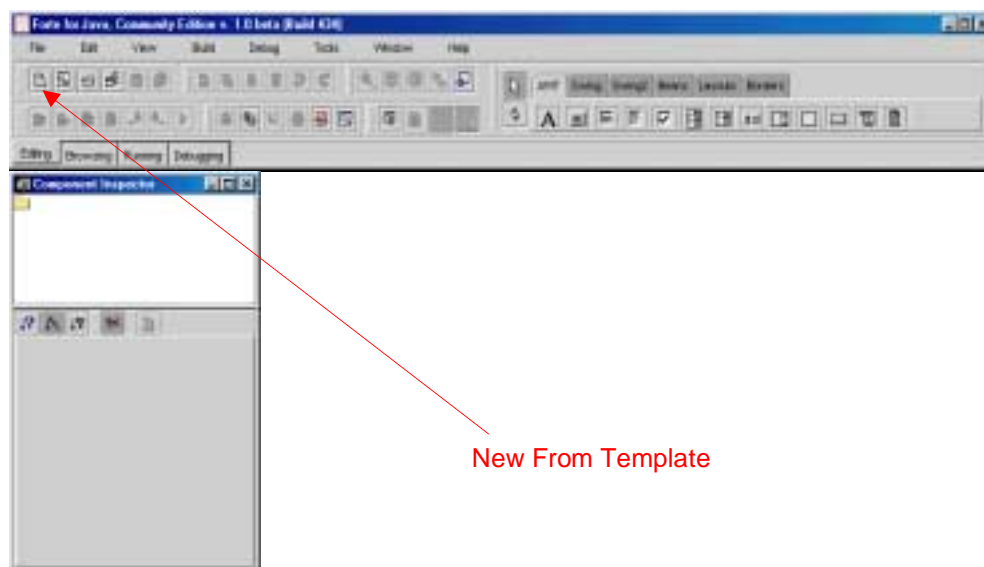
- **Autocode**

Various code templates are available as well as code completion mechanisms.

However Forte does not only support the development of hand coded Java sources. Many standard API classes provided by Java, have a graphical representation in Forte. These representations can be combined via "drag and drop" by using the mouse, while Forte generates the source code automatically in the background. This is a very powerful feature in case for designing user interfaces with Swing and it helps to prevent the typical errors made in hand written code. All programmers who do not have their main interests in programming user interfaces may welcome this kind of support, because it enables them to make GUI-interfaces for their programmes without spending the most time for GUI-programming.

#### 4.2 EXAMPLE APPLICATION DESIGNED WITH FORTE

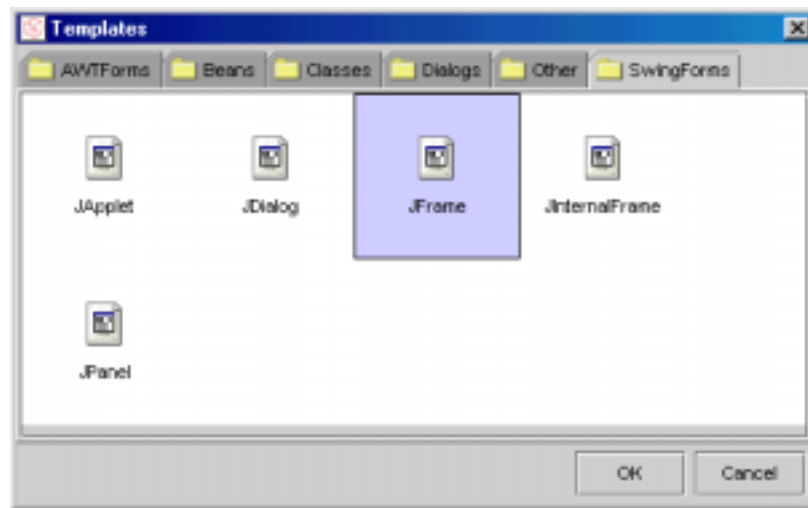
Now the example of section 3.4 will be realised with Forte. After the start of Forte (Figure 4-1) the following windows will appear:



*Figure 4-1: the initial Forte windows*

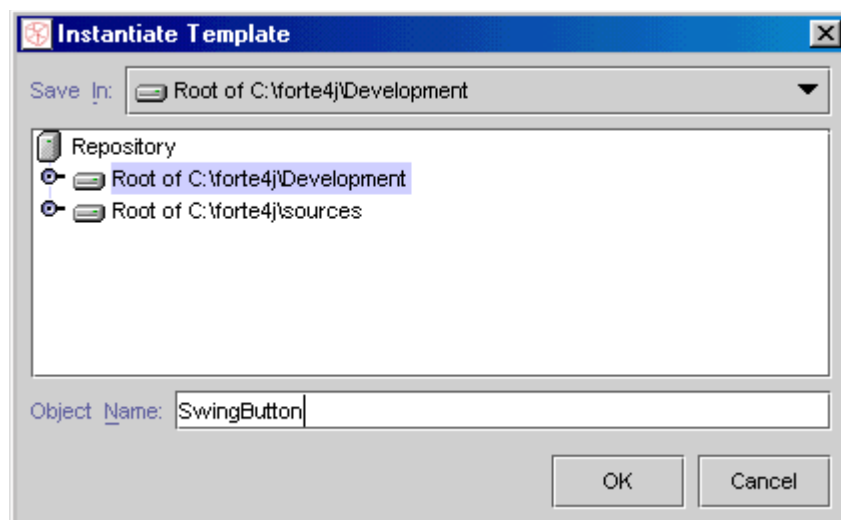
Initially there are two windows the MenuBar and the ComponentSelector. To create a new project the "New From Template" button has to be pressed.





*Figure 4-2: the Template dialog*

The Template dialog appears (Figure 4-2), where the correct pattern has to be chosen. In case of the `SwingButton` example of section 3.4, the `SwingForm` panel needs to be activated and `JFrame` is the right choice. After pressing the OK button a new dialog appears where the directory as well as the new classname must be given (Figure 4-3).



*Figure 4-3: selecting the directory and the classname*

After confirming the input by pressing the OK button the situation changes dramatically, the `ComponentInspector` displays the `JFrame` components, the editor shows the code template for our new class and a graphical representation of the form appears (Figure 4-4).

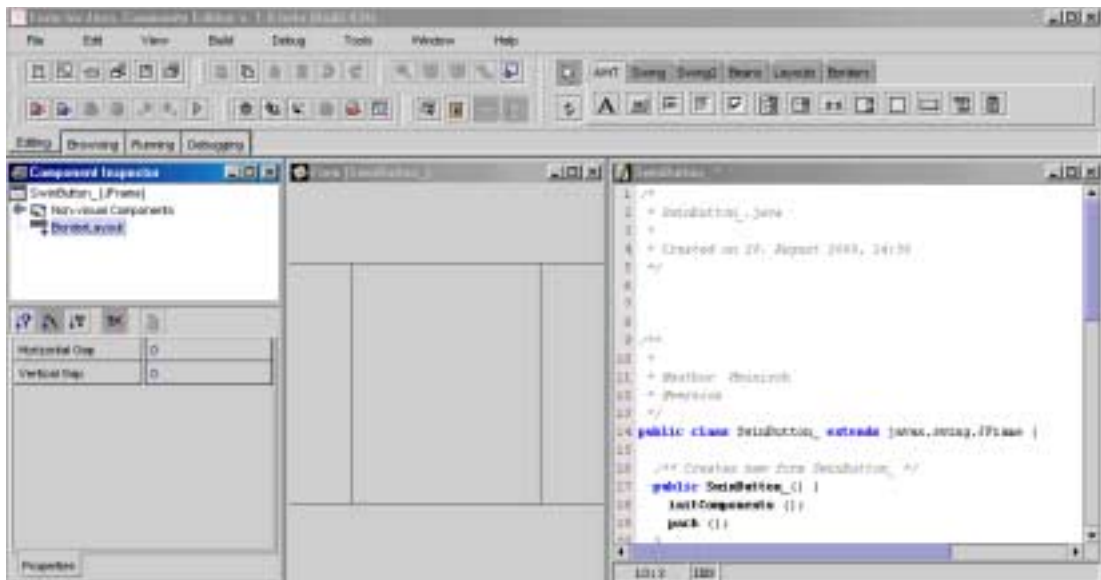


Figure 4-4: the JFrame template exists

Corresponding to the ComponentInspector the default layout is BorderLayout, but as in section 3.4 we want the FlowLayout. To change it, the SwingButton line in the Inspector has to be selected with the right hand mouse button. A pulldown menu appears where the layout can be changed to FlowLayout (Figure 4.5). At the same time the Form window changes its appearance.

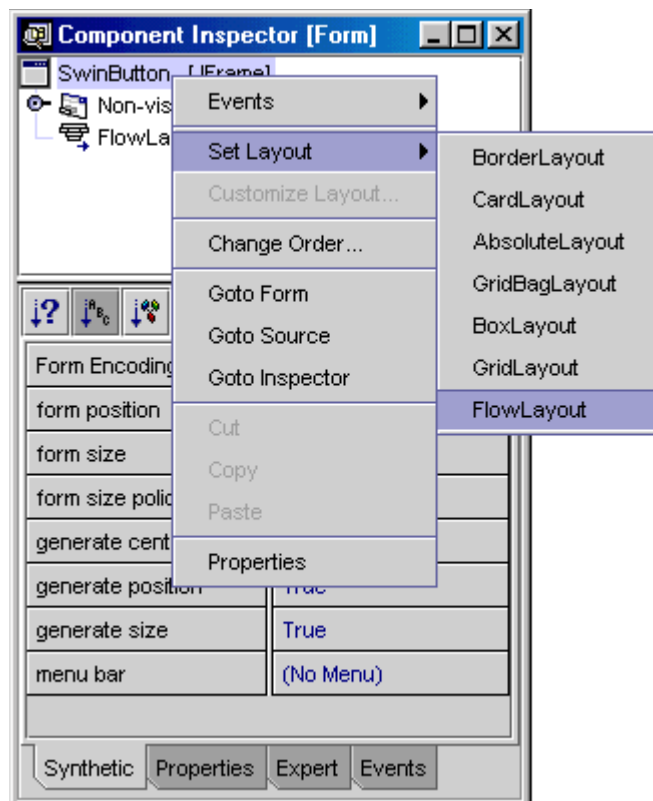


Figure 4-5: changing the layout of SwingButton

The code displayed in the editor has been changed automatically to match the new parameters. If the property tab is activated while the JButton component is highlighted it is possible to change the title property. This property represents the window title and it should be modified to "Demo Application" (Figure 4–6).

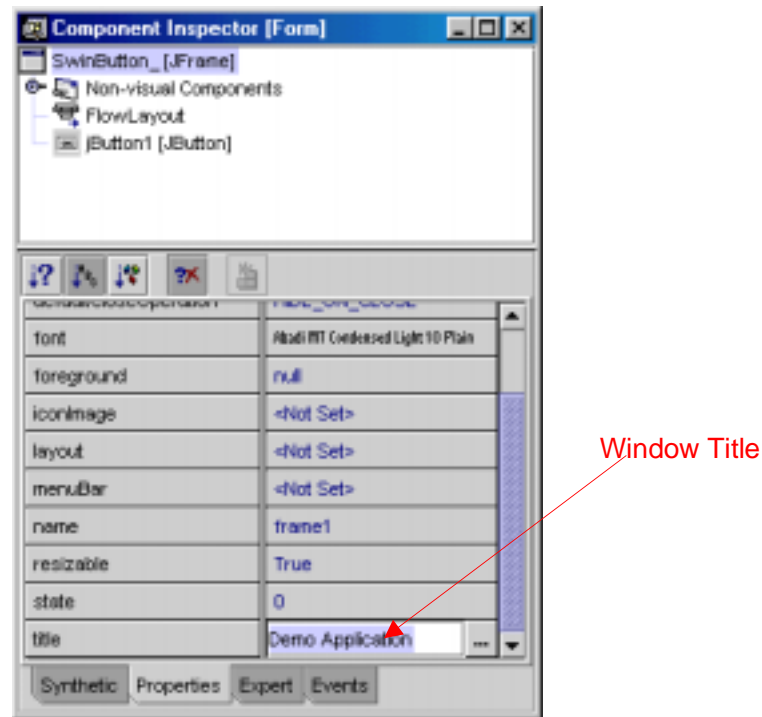


Figure 4–6: changing the window title

In the same way the window size can be set to 200x75 pixels in the synthetic tab into the size property. Now only the button is missing, so this component has to be added. In the right hand top corner of the Menu window the Component Palette can be found, where several components are accessible that can be added via "drag and drop" to the Form window (Figure 4–7). Because a button is needed the Swing tab has to be activated and the button symbol should be pressed by using the left hand mouse button.



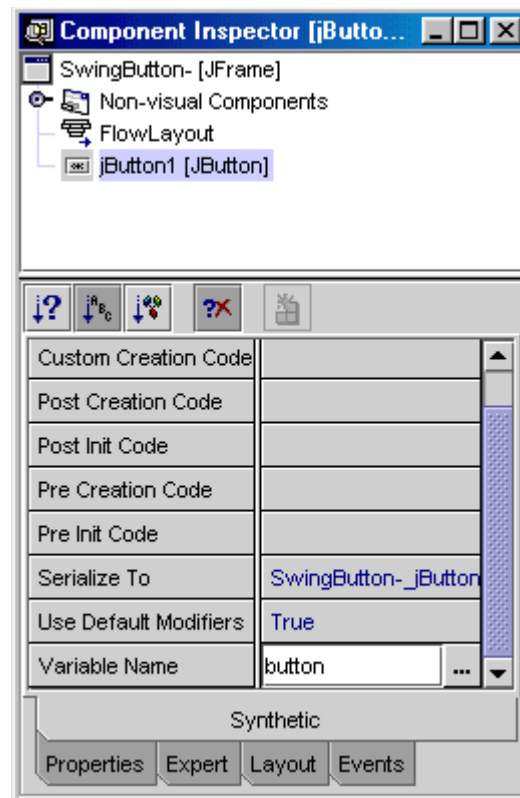
Figure 4–7: Activating the Swing Button

To position the button into the Form window only the cursor has to be moved onto it. One click on the left hand mouse button is enough to put the new component in the right place (Figure 4–8).



*Figure 4–8: the Form–Window with its new button*

In the Component Inspector the name of the reference for the JButton class has to be changed in the Synthetic tab within the "Variable Name" field (Figure 4–9). It is also possible to change the text written in the button by activating the Properties tab where a property "text" can be found. According to section 3.4 the new text is "I am a Swing–button!".



*Figure 4–9: changing the name of the button–reference variable*

Now it is possible to start the program by pressing the start button (Figure 4–10) in the Menu Window. The program just does exactly the same as the one in section 3.4.



Figure 4–10: the Start Button

### 4.3 COMPARISON OF THE SOURCECODE FROM SECTION 2.4 AND 3.2

The code generated by Forte does not look similar to the example code of section 3.4. Nevertheless it has the same function, but there are some conceptional differences that will be discussed in this section. The following programme shows the Forte generated code reduced in size by erasing some automatically generated comments:

```
public class SwingButton extends javax.swing.JFrame {
    /** Creates new form SwingButton */
    public SwingButton() {
        initComponents ();
    }
    /** This method is called from within the constructor to
     * initialize the form.
     */
    private void initComponents () {
        jButton1 = new javax.swing.JButton ();
        getContentPane ().setLayout (new java.awt.FlowLayout ());
        setTitle ("Demo Application");
        setName ("frame1");
        addWindowListener (new java.awt.event.WindowAdapter () {
            public void windowClosing
(java.awt.event.WindowEvent evt){
                exitForm (evt);
            }
        });
        jButton1.setText ("I am a Swing-button!");
        jButton1.addActionListener(new
            java.awt.event.ActionListener(){
                public void actionPerformed (java.awt.event.ActionEvent
                    evt) {
                    jButton1ActionPerformed (evt);
                }
            });
        getContentPane ().add (jButton1);
        java.awt.Dimension screenSize =
            java.awt.Toolkit.getDefaultToolkit().getScreenSize();
        java.awt.Dimension dialogSize = getSize();
        setSize (new java.awt.Dimension (200, 75));
        setLocation((screenSize.width-200)/2,
            (screenSize.height-200)/2);
    }
    private void jButton1ActionPerformed
(java.awt.event.ActionEvent evt) {
        // Add your handling code here:
    }
    /** Exit the Application */
    private void exitForm(java.awt.event.WindowEvent evt) {
        System.exit (0);
    }
}
```

```

    public static void main (String args[]) {
        new SwingButton ().show ();
    }
    private javax.swing.JButton jButton1;
}

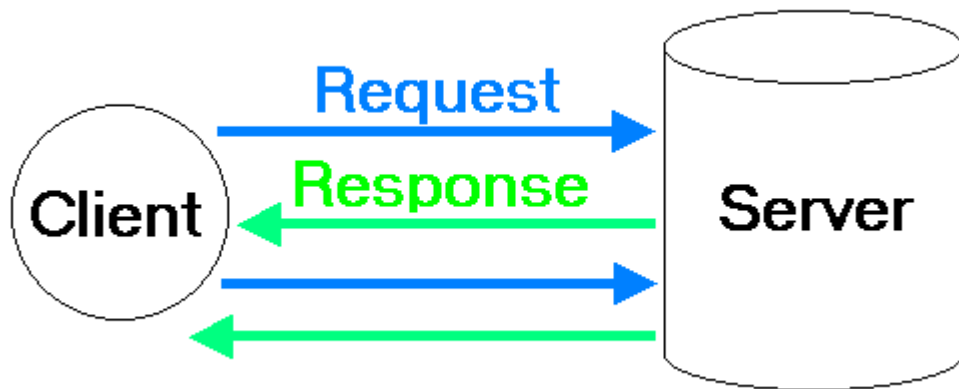
```

The red part of the sourcecode is completely missing in the example of section 3.4. It is a method with the task to react on the "event" when the user presses the button. However this method is empty in this example and that means that it does nothing on activating the button which is exactly the same reaction as in the example of section 3.4. Another difference is, that the programme is split up into 6 methods. Every method has its special purpose for example the `main`-method is the entry point of the program, `exitForm` is the method that is called on exit, `JButton1ActionPerformed` is the eventhandler for the Button, `SwingButton` is the constructor, `InitComponents` initialises all components belonging to the class. This partitioning helps to keep an overview in large projects. The blue coloured text should give a small hot spot to emphasise the basic difference between the two examples. The Forte generated source produces a class "SwingButton" that inherits the class `JFrame`. The use of the `main`-method is only to make an instance of that class and to call the method `show()`. All methods of "SwingButton" do not need an explicit reference to the `JFrame` object, since the invisible "this"-pointer performs the task. Unlike this the example of section 3.4 only works with references to the class `JFrame`.

## 5 The Grasshopper Agent System

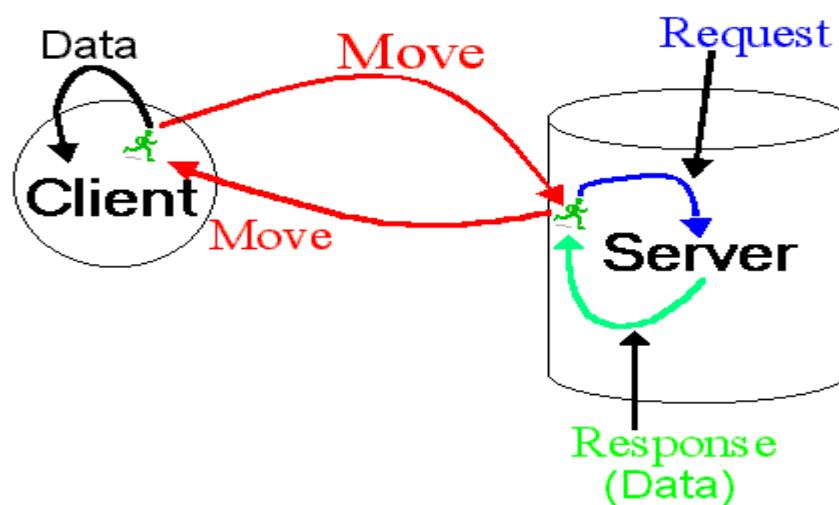
### 5.1 INTRODUCTION TO THE AGENT TECHNOLOGY

Conventional networks are designed according to the "client server" principle. For data exchange between two computers belonging to the same network a physical transmission media is used for example a coaxial cable. The client computer requests data from the server that responds in a more or less long time period. Every request (Figure 5–1) and every response consumes bandwidth of the transmission media. The result is an enormous traffic on the network and a high load on the media.



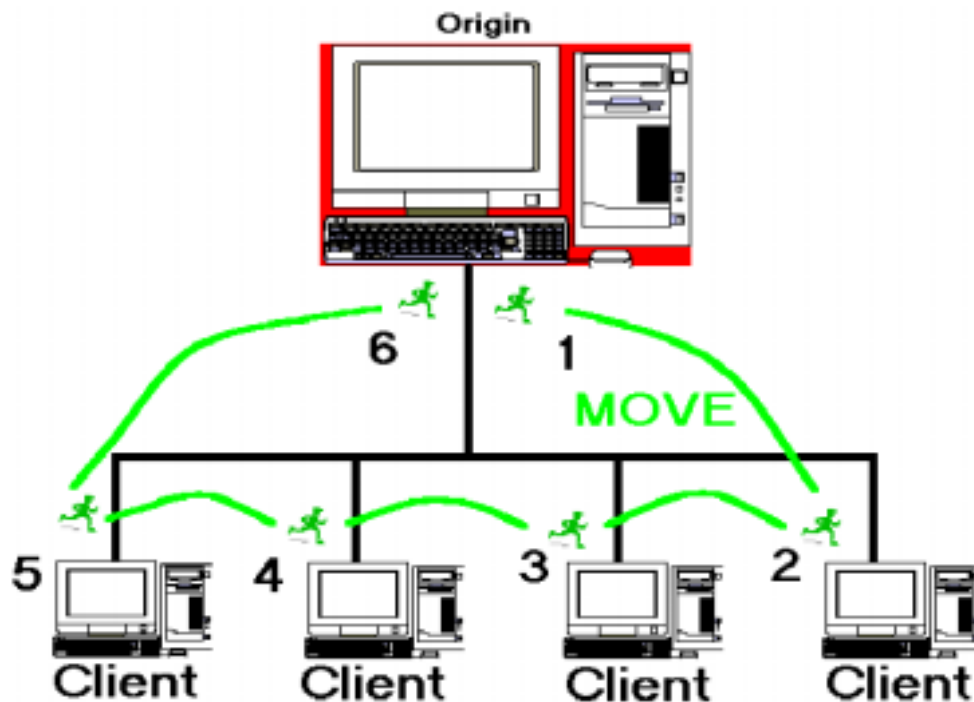
*Figure 5–1 the "client server" communication–principle*

Communication between two or several computers with the use of mobile agents helps to reduce network traffic. An agent is an executable program that is transferred from a client computer to a target computer. After the target computer received the agent it executes it (Figure 5–2). During execution agents have the capability to include or store data. At the end of execution the agent tells the target computer to transfer it, with the included data, back to its source machine. The advantage gained by this technique is, that the data is processed and selected on the target computer and not by the client over the bottleneck of the network. This helps to prevent the transfer of data that is not used by the client. However the traditional client/server architecture will not be replaced by this technology but it may be considered as an expansion of this architecture.



*Figure 5–2 Communication with the use of mobile agents*

Since an agent represents an independent program it is able to provide a new way of communication between several computers. The agent migrates self-sufficiently from its origin-machine to several different computers (Figure 5–3).



*Figure 5–3: an agent "migrates" from computer to computer*

After it has done its work it returns back to the origin. The agent has the capability to transport all the results produced on every visited host like in a rucksack.

## **5.2 WHY USING THE GRASSHOPPER SYSTEM**

Nowadays many agent systems exist, and it is sometimes very difficult to choose an appropriate one. Some of them bases on scripting languages like TCL others are based on common programming languages like C,C++ and Java. The TCL, C and C++ based agent systems are having all one big problem, they do not work on different machines without customisation. It is not possible to execute these agents in an heterogeneous network like the internet, because of the different types of machines (processors, operating systems) that are used there. Also porting an agent system to different machines may be difficult, because of the existing dialects of the used programming languages. Unlike this Java defines a virtual machine with a virtual processor and operating system, that has to be exactly emulated on



every target machine. The advantage of this attempt is, that not the programming languages as well as the processors and operating systems of the target machines have to be standardised. The disadvantage is the slow emulation speed of the existing Java virtual machines. The Grasshopper system is based on Java and it offers the full functionality of the Java-API to the programmer. So it is possible to compile and execute the whole agent system as well as agents on every host that offers a Java virtual machine. This circumstances make it possible to operate an agent in today's Internet, because a Java virtual machine is available for nearly all relevant platforms. An another advantage of Grasshopper is, that a demonstration version is available for free containing very excellent manuals for programming and maintenance the agent system. In addition the company IKV++ that has developed Grasshopper, provides a free of charge E-Mail support also if the demo version is used.

### **5.3 THE GRASSHOPPER DISTRIBUTED AGENT ENVIRONMENT**

The Grasshopper system consists of 4 essential components:

- A single "*Region Registry*" (registration unit of a region)
- Several "*Agencies*" on different hosts
- Several "*Places*" within an agency
- Several "*Agents*" within a place

All these components together are called "Distributed Agent Environment", in the following sections it will be called DAE (Figure 5-4).

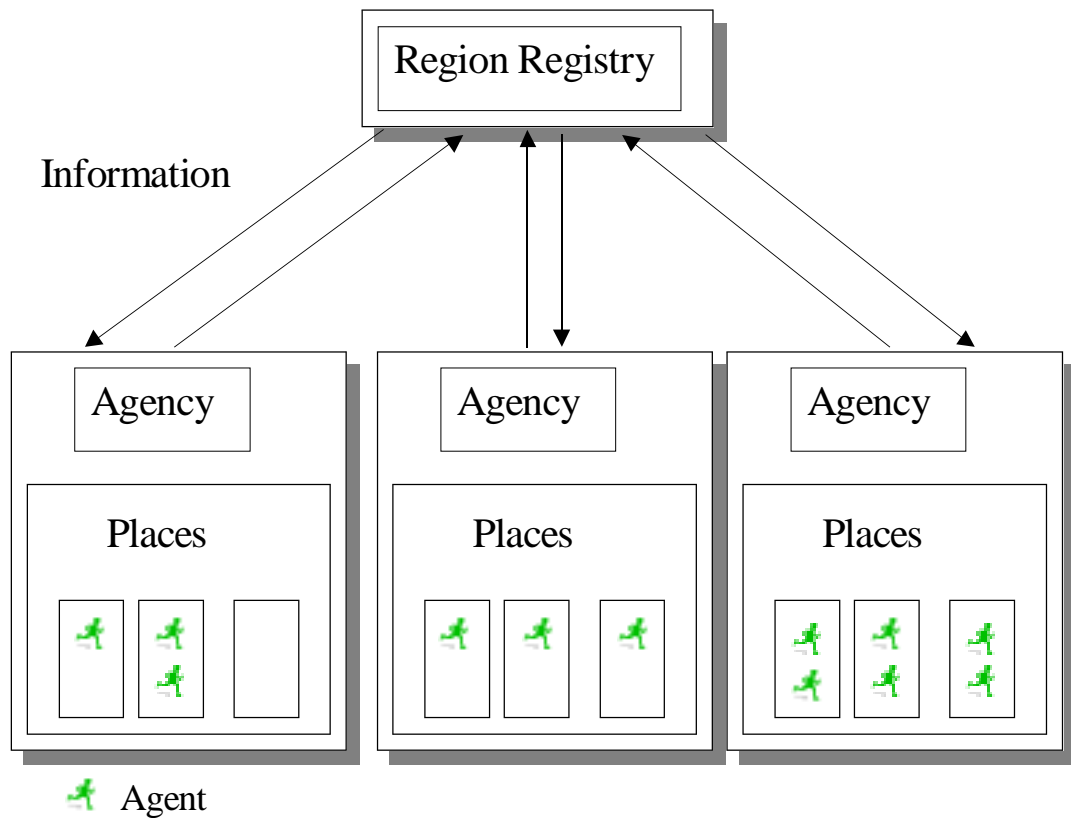


Figure 5–4: the "Distributed Agent Environment"

### The Region Registry

Only one "Region Registry" exists per DAE. The "*Region Registry*" is the centre of a DAE, where all data about the current state of agencies, agents places etc. are stored. Every agent is able to request these stored data and plan its further actions dependent on it. Every agency notifies the Region Registry of any state changes. Such a notification of an agency consists mainly of the following components:

- Name of the agency.
- TCP/IP address and port.
- Name of the available places within the agency.
- Name of every single agent within the places.

A notification is generated if something in the following list happens:

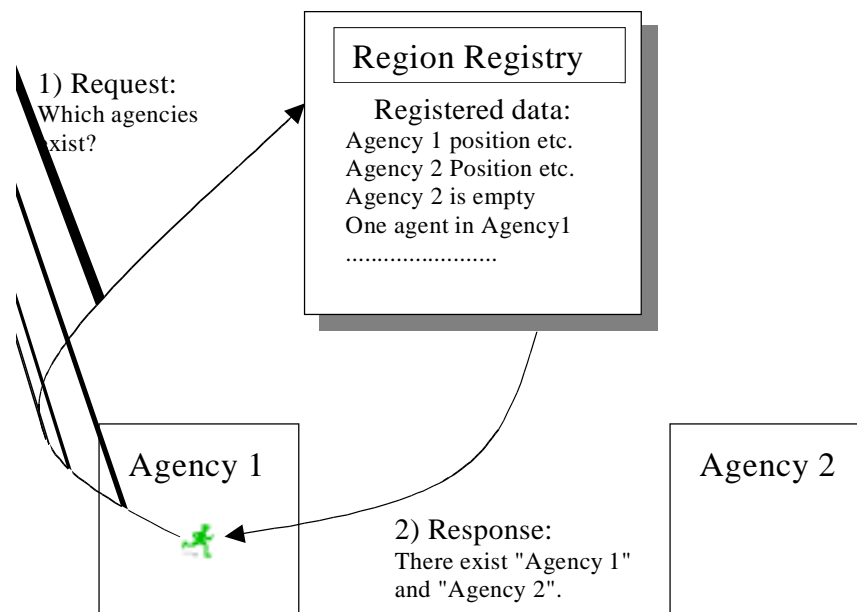
- Deleting or installing of a new place.
- Deleting or installing a new agent.

- Migration of an agent within the agency from one place to another
- Migration of an agent between agencies
- Starting a new agency.

To be able to register at a "Region Registry" every agency has to know the port number, the TCP/IP address as well as the name of the "Region Registry". After an agency has registered, it is an interlocked part of the DAE. Agencies are also able, like agents, to request all information collected by the "Region Registry".

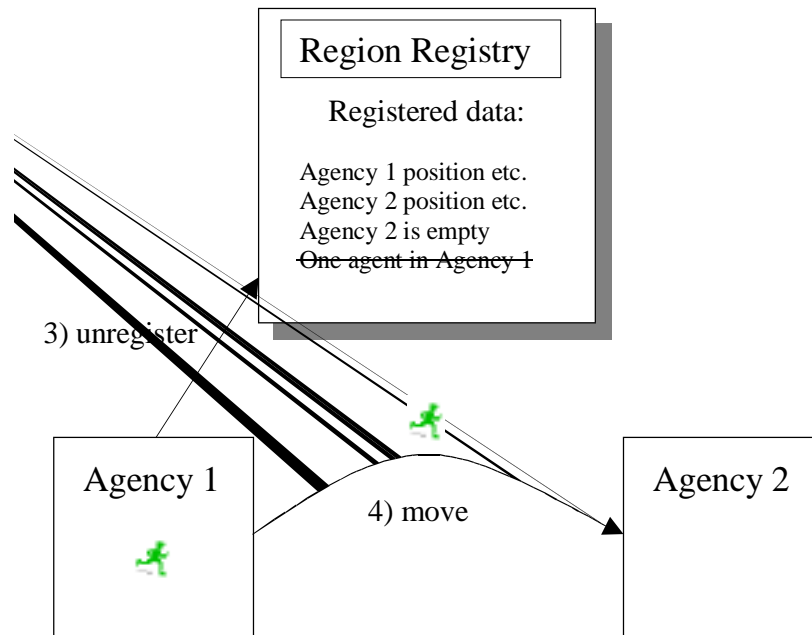
The following figures (5–5 to 5–7) show how a "Region Registry" works during the movement of an agent. In both figures two agencies exist and are registered at the "Region Registry". In one of the two agencies there is an agent, which is also known at the "Region Registry". Places are not shown in these figures, to avoid a confusing view.

- The agent contacts the "Region Registry" to receive information about the existence of all the agencies known in the DAE.



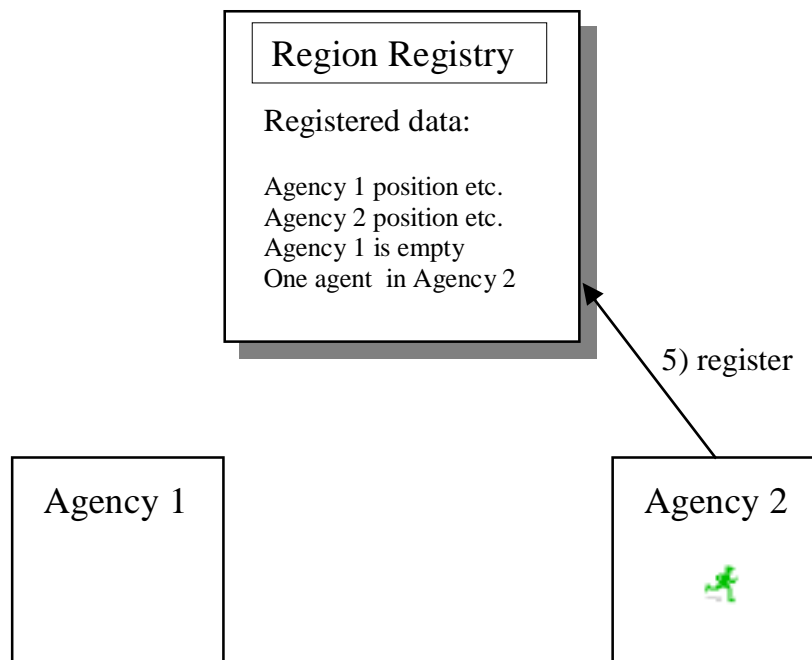
*Figure 5–5: request of the agent*

- The agent on the move from "Agency 1" to "Agency 2".



*Figure 5–6: movement of the agent*

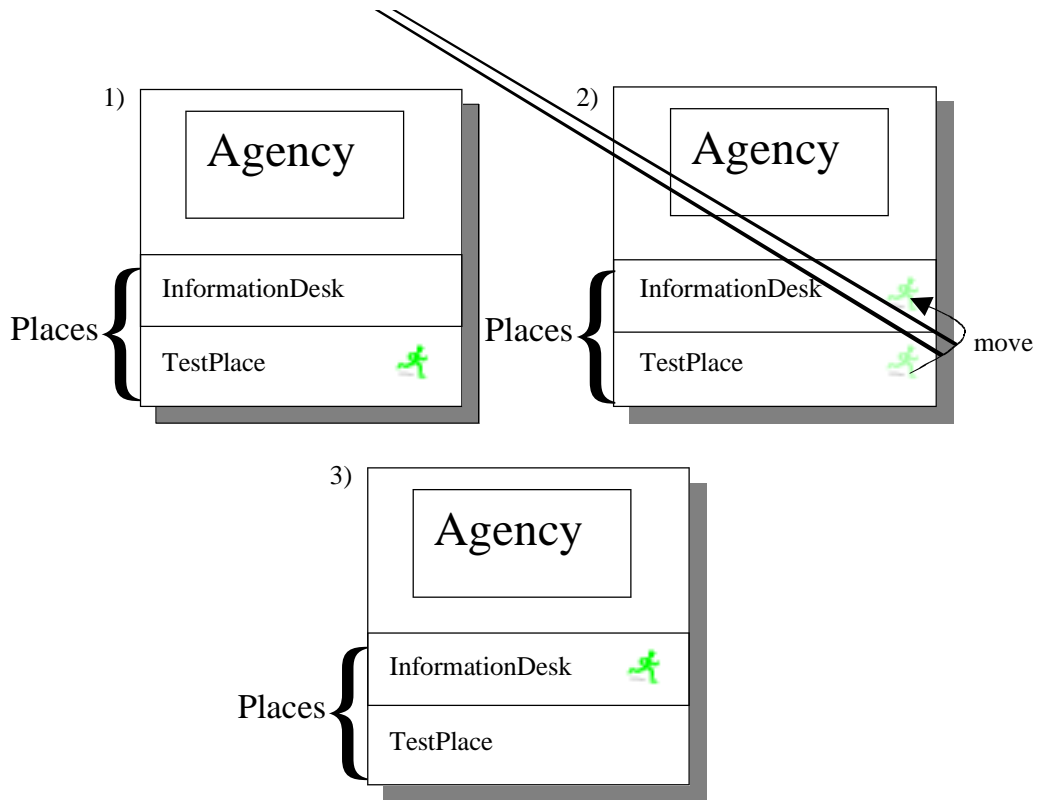
- After the agent has arrived at "Agency 2", the agency notifies the Region Registry about the arrival of the new guest.



*Figure 5–7: arrival of the agent*

**The agency**

An agency is a program which runs on a single host within a network. If an agency registers itself at a region registry it becomes a logical part of the DAE. An agency is subdivided into places, where the agents resides. An agency without places makes no sense, so every agency has at least one default place with the name "InformationDesk". An agent can also move between places inside an agency(Figure 5–8):



*Figure 5–8: movement of an agent from the TestPlace to the InformationDesk*

The agency is the runtime environment for agents like a virtual machine for Java–programs. The agency provides the following services for agents:

### **Place independent inter agent communication**

A synchronous, an asynchronous or a multicast communication is possible, alternatively realised with CORBA or with plain socket Communication.

### **Registration service**

The agency takes care about the automatic registration of the agents residing in it, at the "Region Registry". The agency enables the agent to send requests to the "Region Registry".

## Management service

The user is able to control all agents within an agency by a graphical user interface (GUI). The user can add and remove, start and shut down new agents. Also general information about the agents like names, descriptions, status, class names, codebases are displayed.

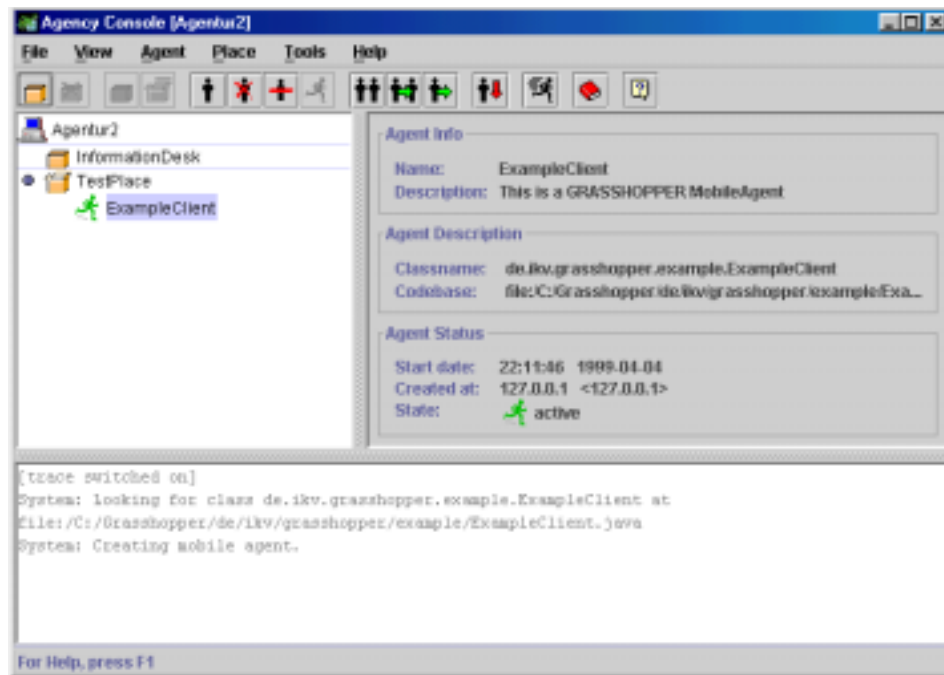



Figure 5–9: the GUI of an agency

In Figure 5–9 the GUI of an agency is shown. The agency owns two places (place–icon: ) , the standard place "InformationDesk" as well as an additional one called "TestPlace". There is an agent in the "TestPlace" with the name ExampleClient. The right half of the agency is used to display various information about the housed agents.

## Transport service

Supports the inter–agency migration of agents, including automatic protection against network errors e.g. cable disruptions, loose contacts etc.

## Security service

The security service is subdivided into two areas:

- External security serves the authentication and checks the data integrity of agents.
- Internal security protects the agencies against harmful or destructive access from the agents.

## Persistence service

The Grasshopper system supports an automatic background storing mechanism that prevents agents as well as agencies from losing data for example if a power failure or a system crash occurs.

## Places

Places can be considered as logical sub-structure of agencies and are used by agents to reside within them. If an agent wants to migrate to another agency it has to know the name of the destination agency as well as the new place inside it. If the agent does not have any information about the structure of places it has only the possibility to migrate to the default place, the "InformationDesk".

## Agents

There exist two kinds of agents in the Grasshopper system, mobile and stationary ones. **Mobile agents** have the ability to move from one physically place in a network to another. Because of this functionality these agents can be considered as an expansion or alternative to the traditional client-server technology. In addition they are able to do some work at their current abode as well as picking up some data and transporting these like in a rucksack. **Stationary agents** however do not have the possibility of migration; they are firmly connected to a place within an agency. Normally they are used as a communication point or server for mobile agents. In principle these agents enlarge the functionality of an agency.

## 6 How to program an Agent

### 6.1 BASICS

The programming of stationary agents is very similar to that of mobile ones. The main difference lies in the father class they are derived from. It is obvious that the father classes differ in their methods, because they have a different focus. The following source code presents a minimal agent programme that only prints out "Hello" in a shell just like the well known program "Hello World":

```
// File: MinimalAgentA.java
package de.ikv.grasshopper.example;
import de.ikv.grasshopper.agency.StationaryAgent;
public class MinimalAgentA extends StationaryAgent
{
    public void live()
    {
        System.out.println("Hello World!");
    }
}
```

Every agent program must implement the "live"-method since it represents the core of an agent. The launch of an agent by the user means that the agency calls the live-method. Unlike ordinary Java programs an agent does not have a main- method. The method "live" performs this task.

After the "live"-method is finished the agent cannot be run again by double clicking on its symbol in the agency. If this behaviour is mandatory the method "action", has to be implemented.

```
// Datei: MinimalAgentB.java
package de.ikv.grasshopper.example;
import de.ikv.grasshopper.agency.StationaryAgent;
public class MinimalAgentB extends StationaryAgent
{
    public void action()
    {
        live();
    }
    public void live()
    {
        System.out.println("Hello World Again!");
    }
}
```

### 6.2 A SIMPLE MOVING AGENT

The following program implements an agent that moves from one agency to another. To get a very simple program the following assumptions are made:

- There are only 2 agencies in the network.
- The second agency has the name "Agency2" and runs on a host with the IP-address 192.168.168.90.



- There is exactly one Region Registry where both agencies are logged in.

The class of the agent now has to be derived from "MobileAgent" which is the father-class of all moveable agents. The red coloured code shows the part of the program that is responsible for the move. Because the `move(...)` command throws an exception it is encapsulated in a try-block.

```
// File: SimpleMove.java
package de.ikv.grasshopper.example;

import de.ikv.grasshopper.agency.*;
import de.ikv.grasshopper.type.*;
import de.ikv.grasshopper.util.*;

public class SimpleMove extends MobileAgent
{
    public void live()
    {
        Location home = null;
        Location newloc = null;
        home = getLocation();
        System.out.println ("Home : " + home.toString());
        newloc=new Location
        ("grasshopperiio:192.168.168.90/Agency2/InformationDesk")
        ;
        try
        {
            System.out.println("Now Moving!");
            move(newloc);
        }
        catch (Exception e)
        {
            System.out.println("I have arrived!");
        }
    }
}
```

If the program works correctly the following output will appear in the start-agency:

```
Home :
grasshopperiio:192.168.168.90/Agency1/InformationDesk
Now Moving!
```

In Agency2 the output will look like:

```
grasshopperiio:192.168.168.90/Agency2/InformationDesk
I have arrived!
```

After the agent has arrived at Agency2 it will be started again there. Because it cannot move to itself an exception is thrown, which is interpreted as the arrival of the agent.

### 6.3 STORING DATA WITHIN AN AGENT

Every private class variable within an agent class remains unchanged in the case of a migration. The Java access modifiers *private*, *protected* and *public* also keep their validity.

```
// File: MinimalAgentC.java
public class MinimalAgentC extends StationaryAgent
{
    private int Wert; //content of this variable remains
                    //in case of a migration
    public void live()
    {
        .....
    }
}
```

### 6.4 MIGRATION RESISTANT CONDITIONS OF AN AGENT

What happens if an agent moves? Well, the agent programme especially the bytecode representing it will be transmitted via the network from one agency to another one that executes the program again from the beginning. Since the programming language Java does not provide a mechanism to store the processor/CPU context or the condition registers of the Virtual Machine, a mobile agent program requires a special construct which conserves the status of the program at the point of time when the agent begins to move. Therefore a simple migration resistant integer variable within the class can be used that will be incremented by the program just before a move. Because of this variable the method `live` can reproduce the conditions desired after the migration. This construct requires to subdivide the method `live` into various "action blocks", that are acting in dependence of the "state variable". The following shortened listing shows this mechanism:

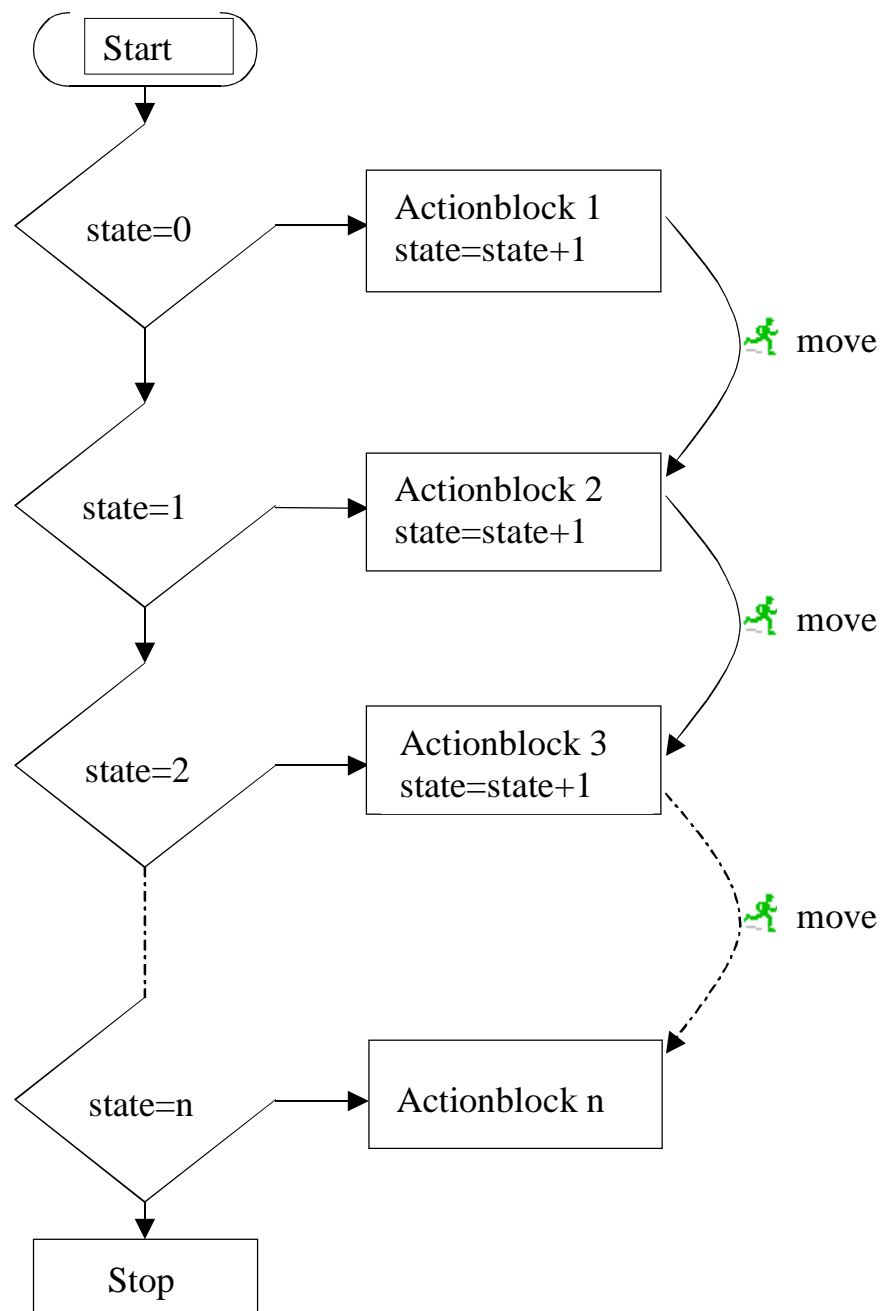
```
// File: MinimalAgentD.java
public class MinimalAgentD extends MobileAgent
{
    private int state = 0; //content remains
    public void live()
    {
        if (state == 0)
        {
            //actionblock 1
            ...
            state++;
            move(...);
        }
        if (state==1)
        {
            // actionblock 2
            ...
            state++;
            move(...);
        }
    }
}
```

```

    if (state == 3)
    {
        // actionblock 3
        ...
        state++;
        move(...);
    }
    .
    .
    .
    if (state == n)
    {
        //last actionblock
        ...
    }
}

```

Figure 6–1 shows a diagram that illustrates the program flow. The green agent symbols indicate a migration of the agent:



*Figure 6–1 The program flow diagram for a migration persistent agent program*

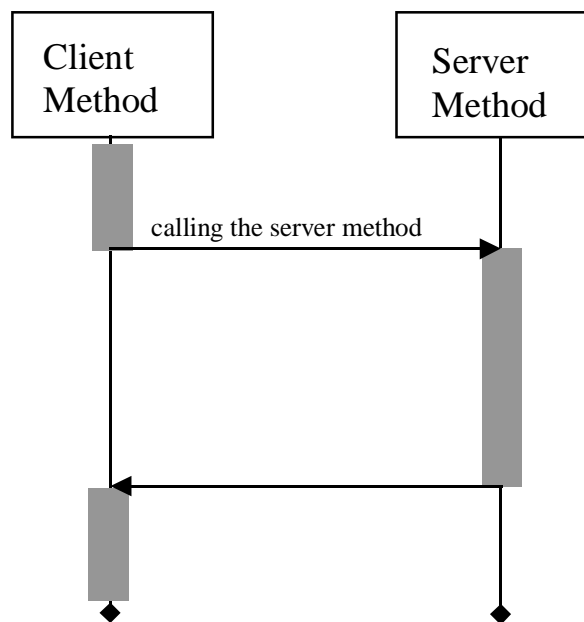
## 6.5 PROXY COMMUNICATION BETWEEN AGENTS

Agents have the possibility to communicate with each other. It does not matter, whether the agents stay within one agency or reside within different agencies. Agents communicate for example to share information they have collected during their migration through the network. There are always two different roles in a single communication, the client as well as the server.

The communication between the agents is carried out by a Proxy (representative) which provides methods for communication like Remote Procedure Call (RPC). There exist four kinds of communication:

### **Synchronous communication**

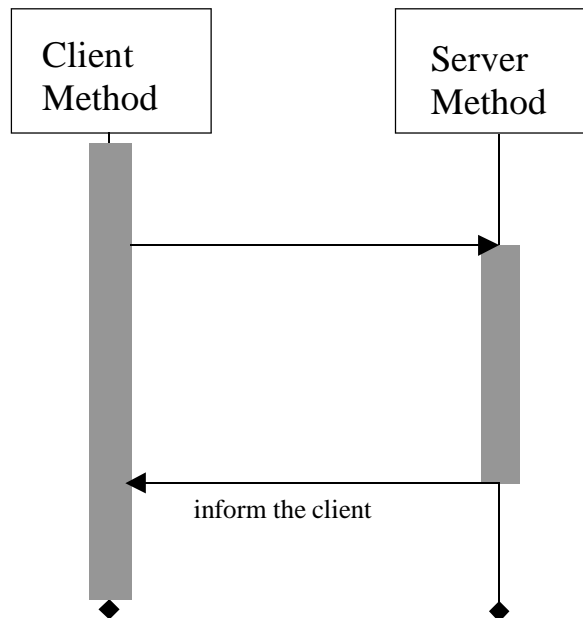
A client calls, via Proxy, a method of the server (Figure 6.2). While the method is executed by the server, the client is blocked. As soon as the execution has ended, the client is unblocked.



*Figure 6–2 Sequence diagram for synchronous communication*

### **Asynchronous communication**

After a client has called a method of a server via Proxy, it will not be blocked. The server just executes it in the background (Figure 6–3). To receive the result of the call, the client has various possibilities. For example it can check the status of the server processing the method by polling. Another more elegant variant is to give the server the possibility to inform the client when the method execution is finished.



*Figure 6–3 Sequence diagram for synchronous communication*

### **Dynamic communication**

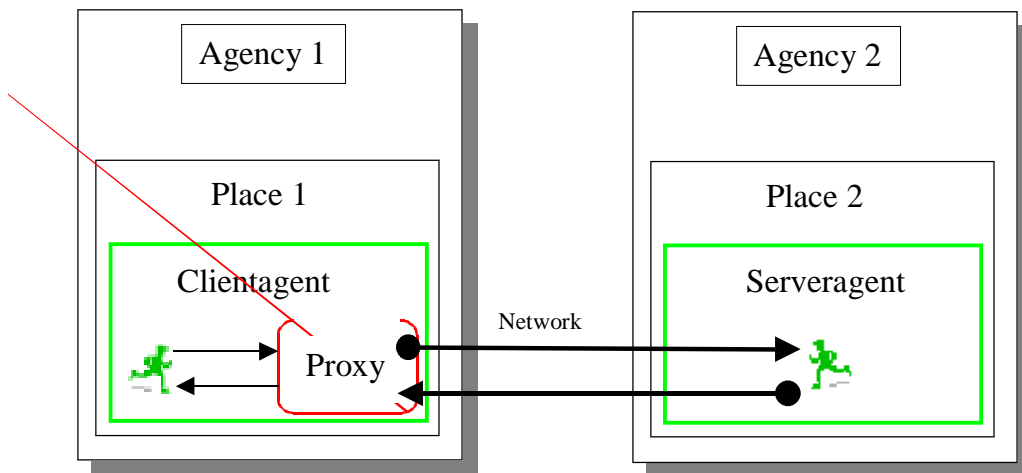
Dynamic communication of agents isn't carried out by a known proxy but via the base/father class of all proxy classes. This mechanism has the advantage that agents can communicate with each other without knowledge about their special proxies. Dynamic communication is a special form of the asynchronous/synchronous communication. The name "dynamic communication" comes from the company IKV++ can be understandable.

### **Multicast communication**

A client has the possibility to make a request to several server agents simultaneously.

### **Implementation of the server side**

Every agent that wants to act as a server has to provide a proxy with an accompanying stub. There exists a tool in the form of a batch script in the Grasshopper system, that generates a proxy-class which can be used by every client agent. The proxy-class encapsulates, invisibly for the client, the whole communication process (Figure 6–4). With this mechanism the client has access to every public-method of the server.



*Figure 6–4 two agents using proxy communication*

For example the following class has to become a new server-agent, which can multiply a given number with 10:

```
// File: ServerAgent.java
package de.ikv.grasshopper.example;
import de.ikv.grasshopper.agency.*;

public class ServerAgent extends StationaryAgent
{
    public void live()
    {
        System.out.println ("ServerAgent is running");
    }
    public String getName()
    {
        return new String ("ServerAgent");
    }
    public int multiply_with_ten (int numbers)
    {
        return numbers * 10;
    }
}
```

If the class `ServerAgent` is compiled with the Java compiler the result is a bytecode-file with the name `"ServerAgent.class"`. The tool that generates the suitable proxy-class is called `"stubgen"`. Its use is very simple, only `"stubgen ServerAgent"` typed in a shell is enough to produce a new Java file `"ServerAgentP.java"` that represents the proxy source code. This file again has to be compiled with the Java compiler so the result is a proxy-class with the name `"ServerAgentP.class"`. Figure 6–5 illustrates the process:

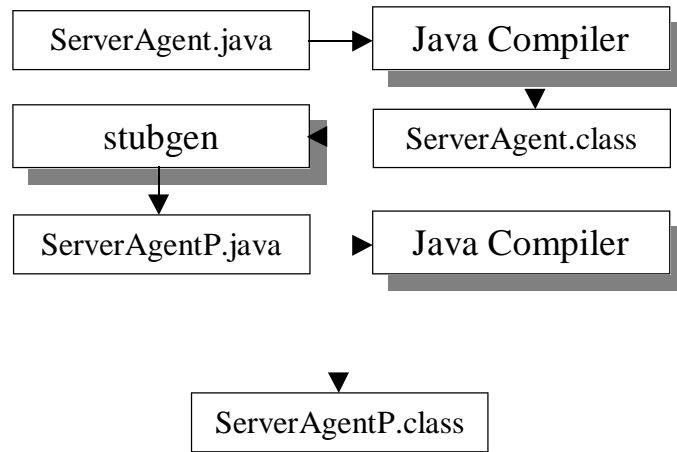


Figure 6–5 generation of a proxy-agent

### Implementation of the client side

One important task for the client is to generate an object of proxy-class "ServerAgentP.class" that enables it to communicate with the server agent. The communication is handled by the proxy from the point of view of the client. The following program shows an implementation of the client, all lines that have to do with the proxy are coloured red:

```

// Datei: ClientAgent.java
package de.ikv.grasshopper.example;
import de.ikv.grasshopper.region.*;
import de.ikv.grasshopper.agency.*;
import de.ikv.grasshopper.type.*;

public class ClientAgent extends MobileAgent
{
    public void live()
    {
        int z = 10;
        String s;
        ServerAgentP SERVERAGENT = null;
        RegionRegistrationP registry = null;
        System.out.println ("ClientAgent alive!");
        ServiceInfo[] agentInfos = new AgentInfo[0];
        agentInfos = listStationaryAgents(); //lookup all agents
                                           //in the agency
        Identifier idown = this.getIdentifier();
                                           //get own identifier
        for (int i = 0; i < agentInfos.length; i++)
        {
            Identifier id = agentInfos[i].getIdentifier();
            // test all found agents
            //only if the identifier
            //differs the proxy will
            //be started

            if (id != idown)
            {
                SERVERAGENT = new ServerAgentP (id.toString());
                break;
            }
        }
        if (SERVERAGENT != null)
        {
            s = SERVERAGENT.getName(); //take the name of the
            //server
            System.out.println ("---> the Servername : "+s);
        }
    }
}
  
```



```

        int value = SERVERAGENT.multiply_with_ten (z);
        //doing some mathematics
        System.out.println ("----> 10*10 = " + value);
    } else System.out.println ("No server present!");
}
}

```

**The output of the client:**

```

ClientAgent alive!
"---> the Servername : ServerAgent
"----> 10*10 = 100

```

The client uses the synchronous communication. Because the other 3 ways of communication work in a similar manner they are not explicitly discussed here.

## 7 THE IDENTIFIED FUNCTIONAL BLOCKS

The functional blocks are identified by comparing the various example agents, provided with the demo version of the Grasshopper system. Listing these examples exceed the size of this document. The following section only shows the results of the examination process.

### 7.1 THE MAIN TEMPLATE

The biggest block is also the one that can be overseen very easy. It is the main-block which represents the framework of an agent. In a normal Java program this framework maybe a class containing the main-method. In the case of Grasshopper Agents it is a class derived from the superclass StationaryAgent or MobileAgent, containing the live-method and the action-method. Not necessary but useful is the method "getName" which returns the name of the Agent and enables the agency to display the name of it. In the case of an mobile agent another useful method is "onMove" which will be the last method executed before a move operation. It can be used to clean up memory or to do other necessary jobs like closing all opened files etc. All red coloured areas in the following source code may differ depending on which kind of agent, stationary or mobile, the user wants to build. The blue, green, magenta, yellow, ochre, grey and black areas are marking areas which may be expanded if the agent gets supplementary functionality.

```
package de.ikv.grasshopper.example;
import de.ikv.grasshopper.region.*;
import de.ikv.grasshopper.agency.*;
import de.ikv.grasshopper.exceptions.*;
import de.ikv.grasshopper.type.*;
import de.ikv.grasshopper.util.*;
import de.ikv.grasshopper.app.util.GOptionPane;
import com.sun.java.swing.*;
import java.util.*;

.....
.....
.....
.....
public class AgentTemplate extends MobileAgent
{
    private Location home = null;           //the home location
    private int state;
    .....
    public void action()
    {
        home = getLocation();
        state = 0;
        live();
    }
    public String getName()
    {
        return new String("(" ..... ");");
    }
    public void onMove()
    {
```

```

} .....
public void live()
{
    .....
    if (state == 0)
    {
        .....
    }
    :
    :
    if (state == laststate)
    {
        .....
    }
}
}
}

```

Definition: There exists at least one actionblock (see section 6.4).

The code-blocks shown in the following sections have the background color corresponding to the area where they belong in the Main-Template.

## 7.2 THE MOVE-BLOCK

The move-block is an essential block for mobile agents. Nevertheless it is not very spectacular. It only makes sense to place it within a grey coloured area of the Main Template. In parallel there exists a list of agencies that should be visited. The reference for this list has to be placed in the ochre part of the main-template. The list will be generated within the agency info block (see section 7.3).

The following code shows the move-block the red coloured part will be explained in section 7.3:

```

Location destination=getNextLocation();
try
{
    move(destination);
}
catch (Exception e)
{
    System.out.println("Error can not move!");
}

```

## 7.3 THE AGENCY/PLACE INFO BLOCK

This block is divided into two parts, a Grasshopper specific one and a non specific one. The specific part is to request the Region Registry for a list of available agencies. The non-specific part is to select the agencies the user wants the agent to visit. For example the user only wants the agent to visit

agencies containing places "car sale" which indicate that there is an stationary agent that offers cars. These and many other constraints can be imagined and should be adjustable within the GUI of the agent designer. In the move-block the `getNextLocation()` -method will inform the agent about the next agency it has to visit. The following source code shows the specific part of the block, all non-specific parts are not developed explicitly in this document.

```
ServiceInfo[] agentSystems = new ServiceInfo[0];
PlaceInfo[] places = new PlaceInfo[0];

RegionRegistrationP registry=null;
try
{
    registry=new RegionRegistrationP
        ("de.ikv.grasshopper.region.RegionRegistration",
System.getProperty("grasshopper.region.registry.address"));
}
catch (Exception e)
{
    System.out.println("Error: contacting Region
Registry!");
}
SearchConstraints constraints = new SearchConstraints();
ServiceInfo agentSystemsSpecified =
    constraints.createServiceInfo();

agentSystems = registry.lookupAgencies
    (agentSystemsSpecified);
    // in this list now all agencies are stored

places = registry.lookupPlaces
    (constraints.createPlaceInfo());
    // in this list now all places are stored
```

## 7.4 THE AGENT INFO BLOCK

If an agent wants to find out which other mobile agents exist in the system it needs the following block:

```
AgentInfo[] agents = new AgentInfo[0];

SearchConstraints constraints = new SearchConstraints();
agents =
registry.lookupMobileAgents(constraints.createAgentInfo());
```

The agent info block for stationary agents looks very similar so it is not shown here. As in section 7.3 the agent-list pointed to the `agents` reference has also to be selected according to constraints given by the user.

## 7.5 COMMUNICATION BLOCKS

Communication between agents is a useful feature for agents. Grasshopper offers the full range of communication over proxies like the well-known remote procedure call (RPC). This range is too powerful for an easy-to-use GUI designer. So the communication possibilities have to be reduced to achieve an easier usage. If we consider the view of a "standard" non-technical oriented user he or she only wants his or her agent to have access to an easy structured database like a table or spreadsheet. If the agents just store their data in such a way, the access can be easily generalised. The following table is an example for the content of an agent's database:

<i>Car</i>	<i>Model</i>	<i>Colour</i>	<i>Price</i>
Rover	65	Green	10000
Jaguar	X11	Blue	100000
Audi	80	Yellow	5000
Ford	Sierra	Lila	6000

Every agent owns a method that returns the blue coloured headline. Another agent can decide from this information if the content is of interest to it or not. If it is, another method has to deliver one complete row. For example, if the agent wants to get all cars it will receive something like Rover, Jaguar, Audi, Ford. This simple model demands 2 functions accessible via proxy as well as a simple database reference.

```
Database data = new Database[0];
                // reference to the database

public String[] getHeadline()
{
    // communication function
    .
    .
    .
}
public String[] getRow(String row)
{
    // communication function
    .
    .
    .
}
```

An agent normally wants to communicate with other ones if it is in the same agency. The agent info block of section 7.4 delivers a list of all possible agents. This list is successive processed. The following source code illustrates such a block, the red coloured lines indicate that instead of Java code, plaintext is used to describe the functionality.

```

for (i=0;i<agents.length;i++)
{
    make an instanze of the proxy of agent[i]

    String s=getHeadline();

    select the received information

    String t=getRow();

    Select the wanted information and put it in the local
    database
}

```

The class `Database` will not explicitly be developed in this document. However it provides the infrastructure for a simple database. How to get a proxy for these methods is described in section 6.5.

## 7.6 ADDITIONAL NON GRASSHOPPER SPECIFIC BLOCKS

Only the working principles of the following blocks are described.

### The agency/place selector block

This block decides which agency or place to keep in the list of agencies. The user should be able to select agencies by name with wildcards. For example the constraint `car*` selects only agencies like `car`, `carpet` etc. In addition a selection by IP-Address, existing places, hosted agents should be possible.

### The agent selector block

This block selects the results of the agent info block. The agents may be selected by their name (with wildcards), by their agency, by place etc. This is useful if the agent wants to communicate only with a selected choice (see section 6.5).

### The database block

This block provides the needed database functionality for section 6.5 as well as the general database support for the agent. The user also has the possibility to predefine the content of the database with the graphical designer for agents.

### The database selection block

This block includes the selection of the headline as well as a selection of the values of the rows. For example if the agent got a row of car prices he may select only the columns with values between £2000 and £5000.

### Conditional jumps

Conditional jumps are very useful for inexperienced users, instead of *"do-while"* or *"repeat-until"* constructs. The GUI designer has to provide something that looks like a jump. This item or graphical element has to be translated into a *"repeat-until"* construct or something similar.

## **8 Design of the graphical user interface**

### **8.1 ADVANTAGES AND RISKS OF GRAPHICAL USER INTERFACES**

#### **Advantages:**

Well designed graphical user interfaces simplify the human machine interactions. They reduce the memory requirements imposed by the user and allow association with the real world. Optical reception is one of the best developed skills nature gave to humans. Symbols can be recognised faster than text, shape and colour are excellent usable for classifying objects, elements or status. The use of symbols aids learning and makes it easier for people to solve their problems. More concrete thinking helps to avoid errors.

#### **Risks**

If a system is not designed well, the above opportunities can be changed to the opposite. If the meaning of icons is not clear or if the shape of two icons used in a completely different context is too similar users become confused. If associations made by the user lead to errors or to non– resolvable problems the result is confusion, frustration, aggression and, as an effect, a lower productivity. Also the human comprehension limitations need to be taken into account., for example the number of icons a user can remember is limited, sometimes it is better to use plain text in addition to a graphical representation.

### **8.2 GENERAL DESIGN GUIDELINES**

A graphical system consists of objects and actions. Objects are the elements a user can see on the screen, and can be manipulated as a single unit. A well–designed system keeps the focus of the user on the objects and not on how to carry out actions. The graphical interface has to act as a connector between human and machine as well as a separator. The connector has to provide the full access to the computation power of the machine, the separator has to minimise the possibility of damage caused, by handling errors.

To get a good design the following principles should be taken into account:

#### **Components**

A system should not confuse a user, all components should:

- Have the same look and feel.



- The same action should always generate the same result.
- The function of elements should not change.
- A clear and well grouped arrangement.

### **Design consistency**

Design consistency is the most important rule of all design activities. Consistency helps to reduce requirements for human learning, because it allows the transfer of skills learned in one situation to another.

### **Clarity**

All functions, visual elements, metaphors etc. should be conceptually clear. Visual elements should be understandable, relating to the users real world.

### **Directness**

Actions should only result from user requests, performed immediately. Every element of the GUI should have a special functionality and provide, if activated, a quick response.

### **Control**

The user should have the feeling that he or she controls the machine and not the vice versa.

### **Efficiency**

Hand and eye movements should be avoided to a minimum. Frequent change of the input devices such as keyboard and mouse slows the productivity down.

### **Flexibility**

A system should not only be designed for one special group of users. For example a GUI has a mouse as a standard input device, but it provides shortcuts for the keyboard. An advanced user may tend to use the fast shortcuts instead of the slow mouse.

### **Familiarity**

The interface should use concepts that are familiar to the user. For example the pull down menu has to be on the top of a window not on the bottom. Also associations with the real world should be possible.

### **Fault-tolerance**

It lies in human nature to make errors. People make mistakes, a system should tolerate this and give plain text information of what went wrong.

### **Predictability**

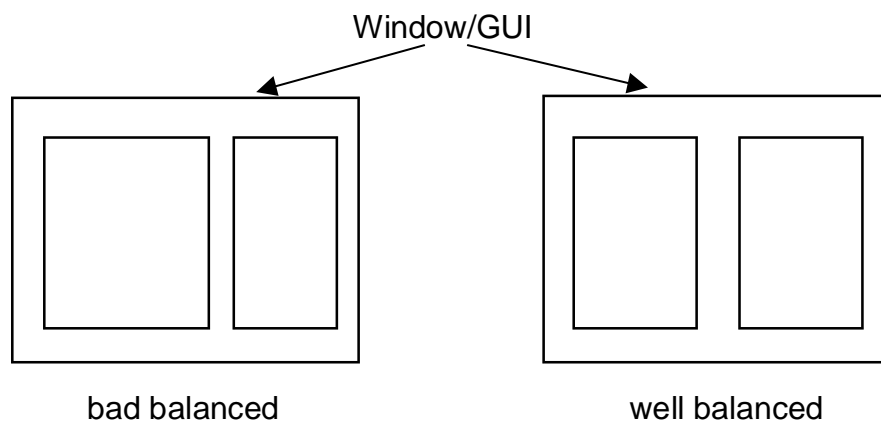
The reaction to a user action should always be clear. This should also be the case, if an error was produced by the user.

### **Simplicity**

Complex dialogues, too many options etc. should be avoided. Also a clear layout of the window, understandable icons, logical functional groups etc. helps to keep the interaction simple.

### **Design Balance**

Humans desire a equilibrium in design. Vertical and horizontal placement of elements needs to be balanced (Figure 8–1):

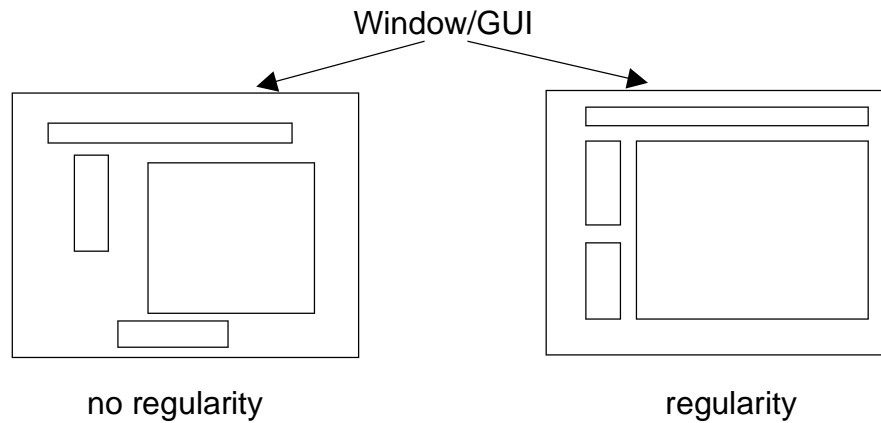


*Figure 8–1 Well/bad balanced GUI*

Not all applications allow a 100% balanced design, but it is the goal for the design to be as balanced as possible.

### **Proper placement**

The screen should be separated in clearly distinct areas. Regularity of design is the goal (Figure 8–2).

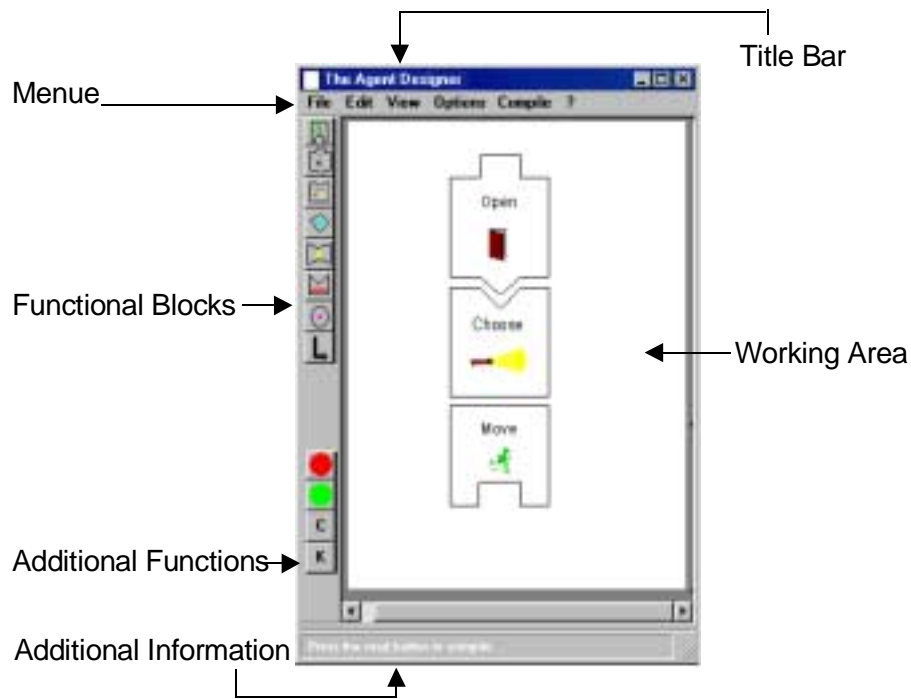


*Figure 8–2 Proper placement of components*

To develop a perfect or nearly perfect GUI many more rules have to be taken into account. Also intensive usability tests with users have to be made. For this purpose Forte is a very useful tool because it allows rapid prototyping. Forte offers the possibility to design a GUI very fast with all elements needed. By default all elements get a minimal "response functionality". For example if a button is pressed it moves down etc. With such a prototype, intensive user tests can be carried out. Dependent on these tests the design can be successively improved. However all these procedures are beyond the scope of this project. The following section will only give an idea of the design of the agent designer GUI.

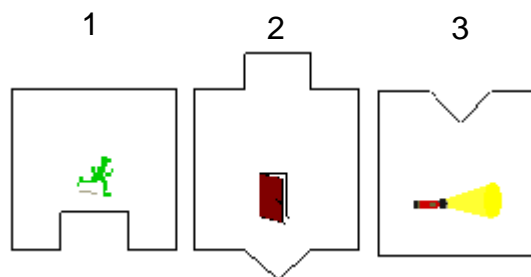
### **8.3 THE GUI FOR THE AGENT DESIGNER**

If fast results are claimed it is helpful to copy the design of already existing graphical interfaces. For the agent designer, the simple and familiar "Microsoft Paint" program is used like a blueprint. Figure 8–3 shows a prototype of the graphical user interface. It only shows the principal appearance, there is still room left for optimisation:



*Figure 8-3 The GUI of the designer*

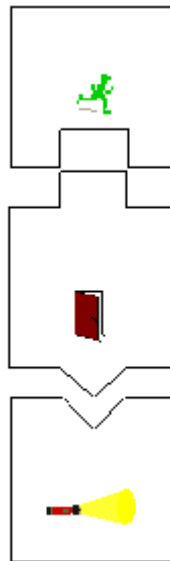
The main idea is that the user can place a functional block in the working area via drag and drop from the "Functional block area". Because every functional block has its own special shape it can only be connected to a another matching one. The system is very similar to a puzzle or to Lego. It prevents the user from grouping the blocks in the wrong way. For example if there exist 3 kinds of functional blocks (Figure 8-4):



*Figure 8-4 Three examples "functional blocks"*

It is obvious that these three blocks can only be combined in one special way, other combinations are not possible. This system is chosen to prevent errors as well as to support the user's ability of association. It helps the user to transfer knowledge of the real world (Puzzle etc.) to the software model. The demanded structure of an agent program is now represented in an easy

to learn graphical way, that any non–technically oriented user is able to understand without knowledge of agents or Java programming (Figure 8–5).



*Figure 8–5 The correct combination*

In the next section some selection blocks are introduced. Their task is to select information received by information blocks. The user is able to define the selection parameters. By double clicking any selection block a dialog has to appear which offers some text fields or buttons to parameterise the selection.

#### **8.4 FINDING A GRAPHICAL REPRESENTATION**

##### Dependency Table

All blocks have minimum two dependencies:

1. Which block may be an appropriate successor.
2. Which block is a appropriate predecessor.

These dependencies are shown in the following table. The first dependency is represented by the rows, the second one is shown in the columns. To save place some shortcuts are used:

Agency/Place Info Block	=	API
Agent Info Block	=	AGI
Agency/Place Selector Block	=	APS
Agent Selector Block	=	AGS
Database Block	=	DB
Database Selection Block	=	DBS
Move Block	=	MOV

First Block of State = FBS  
Last Block of State = LBS

	API	AGI	AGS	APS	DB	DBS	MOV	LBS
API	–	⑦	–	⑦	–	–	⑦	–
AGI	–	–	⑦	–	–	⑦	⑦	–
AGS	–	–	–	–	–	⑦	–	–
APS	–	⑦	–	–	–	–	⑦	–
DB	–	–	–	–	–	–	–	⑦
DBS	–	–	–	–	–	⑦	⑦	–
MOV	–	–	–	–	–	–	–	⑦
FBS	*	–	–	–	⑦	⑦	–	–

*Table 8–1 The dependency table*

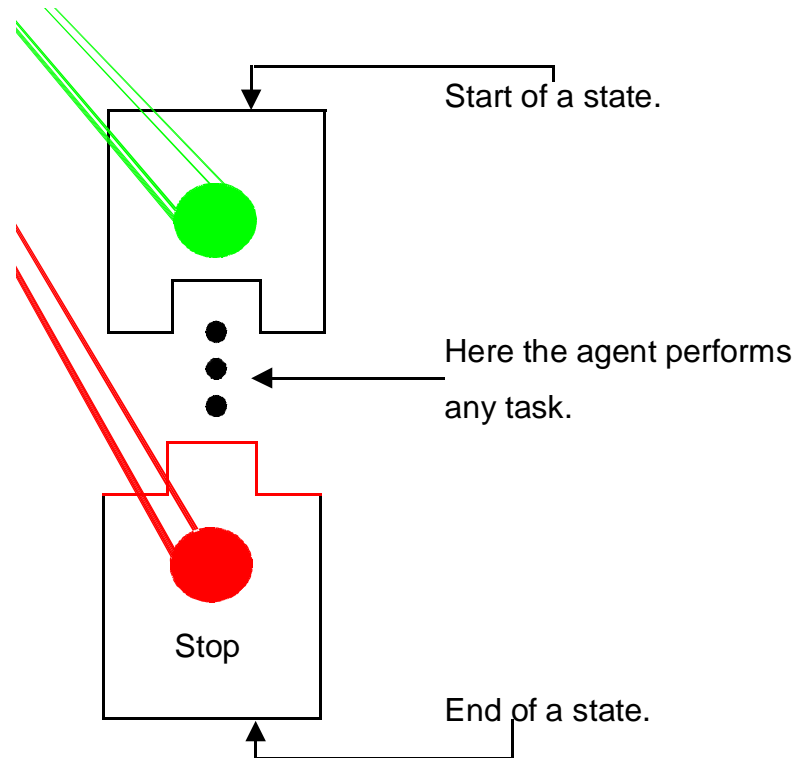
Some interpretations of the table:

In the table (Table 8–1), the first line shows that API may be followed by AGI, APS and MOV. MOV appears only at the end of a state, because MOV increments the static state variable and the execution host changes. DBS is a special kind of block, it exists only after the top of a state and is followed by the end of a state. That means that one state has to be reserved for the DB block. The DBS block is the only one that may be repeated n–times. This is possible because there may be more database requests needed in a single state. If two points appear among each other, like in the MOV column at the API and AGI line, the design of the bottom of the API and AGI block has to be equal, or the MOV block has to offer two designs for its top line. If two points are situated side by side the tops of the following blocks have to be similar or the previous block has to offer two different bottom lines.

Corresponding to the table the graphical representation of the single blocks will be developed in the following.

## States

Functional blocks are encapsulated in states which is described in section 6.4. Every state has a distinct start and endpoint, represented by functional blocks with no connection area at the top or at the bottom. This may be an optional appearance provided by some special blocks. Figure 8–6 shows the principal appearance of these elements. To provide a good overview the blocks in between are omitted.



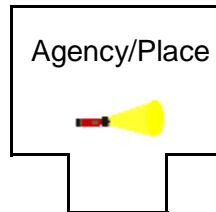
*Figure 8–6 Start/End of a state*

## The Main Template

The Main Template has no explicit symbol representing it. If the GUI is started by the user the working area will be automatically started. This area can be considered as the graphical representation of the Main Template, because it provides the background or base for the other functional blocks.

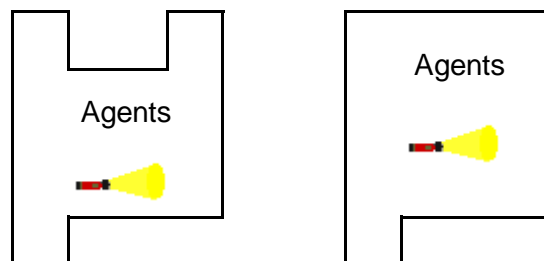
### The Agency/Place Info Block

This block normally is one of the first used by an agent, because it has to know the migration possibilities. It is only possible to place this block at the top of a state.



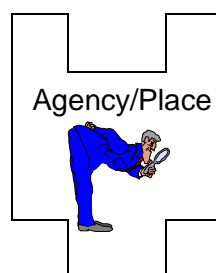
### The Agent Info Block

This block provides a list of all agents. To simplify the design it can only appear after the Agency/Place Info block.



### The Agency/Place Selector block

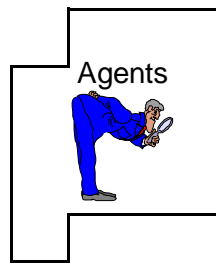
The selector block follows the Agency/Place block. If it does not appear after an Agency/place block all agencies are selected.



### The Agent Selector block

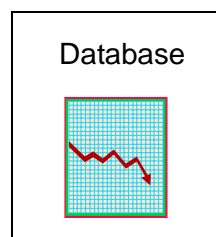
After an Agent Info block has identified the agents, they may be selected.





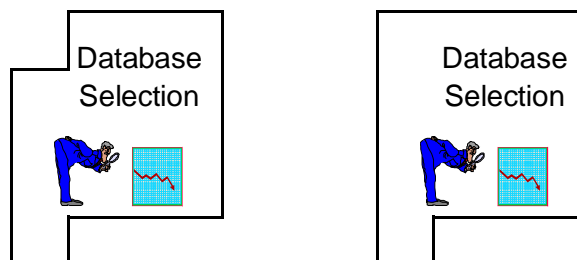
### The Database block

This block produces an instance of a database. Because it does not follow any other block, and no block follows it directly the shape is a simple square.



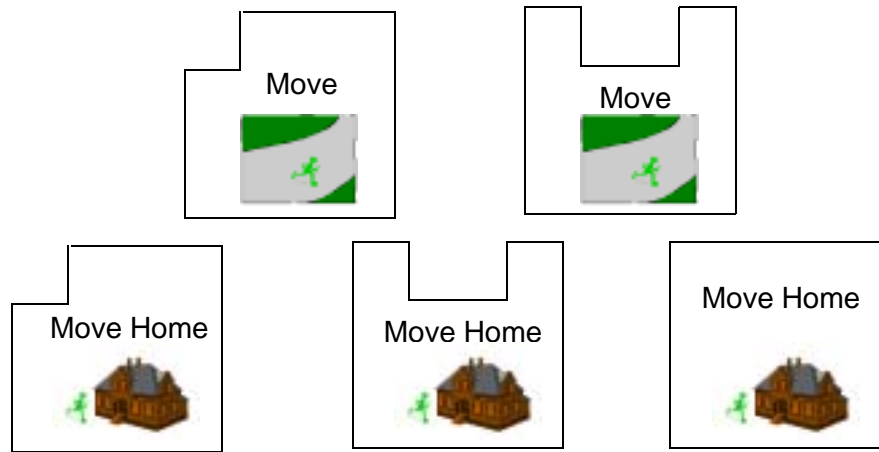
### The Database Selection Block

This block is used to select the content of a remote Database block via proxy (see section 7.5). This selector block has two different designs.



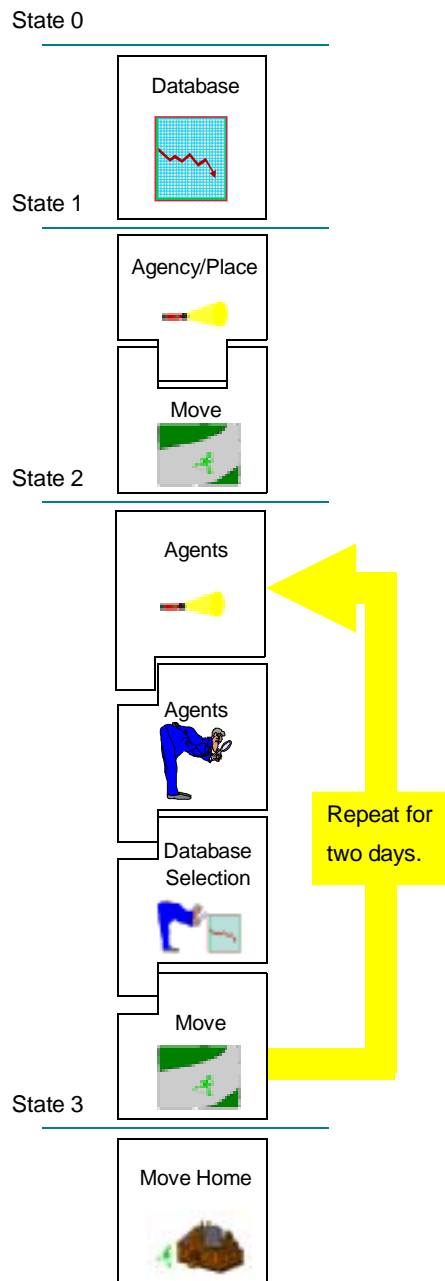
### The Move block

The Move block enables the agent to move. The Move Block offers 2 standard interfaces, because it can follow API, AGI, APS and DBS. There also has to exist a special Move blocks called "Move Home" which forces the agent to move to its home agency.



## 8.5 THE REPEAT UNTIL CONSTRUCT

If an agent has to visit 1000 or more agencies it would be very exhausting for the user to design  $1000 + n$  states. What he or she will need is a "repeat until" construct (Figure 8–7). As a graphical representation for that purpose a yellow arrow is chosen:



*Figure 8–7 A graphical representation of an agent programme*

Figure 8–7 shows an agent programme that initialises a database (state 0) and moves to the first agency (state1). At the first agency all agents are chosen for communication and all their databases are tested to determine if they contain something interesting and in case of a hit the data is stored in

the local database. Once all databases are tested it moves to the next agency. It will do this for two days in the network (provided that the number of agencies available is sufficient). After two days it will go back to its home agency. For simplification the repeat until constructs are only possible from the end of a state to the beginning of the same or a previous state. The conditions determining when to stop looping have to be requested by appropriate, simple dialogs. These dialogs should appear automatically after a user has placed the yellow arrow.

## **8.6 ANNOTATION TO THE DESIGN**

The design of the functional blocks does not completely prevent user errors. For example if in an agent like in figure 8.6 the user has forgotten the Database Block the agent would not work correct, because it can not store the data received by the Database Selection Block. In such a case the interface designer has to inform the user that something is missing, during the process of compilation. This means that the interface designer has to know the logical relations between the functional blocks. These relations can easy programmed with relation tables.

## **9 Conclusion And Further Work**

### **9.1 CONCLUSION**

As shown in this dissertation it is possible to develop an graphical agent designer for the Grasshopper agent system. One of the goals was to find an graphical representation for the functional blocks, that prevents the user to combine them in a wrong way. This goal was not achieved completely:

- Some functional blocks need different shapes depending on their predecessor.
- Some functional blocks are stand alone types like the Database Block. A user is able to position it for example at the last state of an agent, instead of the first.
- Repeat until constructs may be without sense, for example they will never stop looping.

It is the task of the compiler or agent generator of the agent designer to tell these three kinds of errors the user. Maybe a “quiet” background compiler is able to identify these errors early and inform about them in an extra window.

The design of a GUI is a really difficult task. Tools like Forte supports the developer considerably, also in case of rapid prototype design. But to make a real good GUI some experimentees of the target user group should be chosen to test the GUI. These tests will help to improve the design. Because of the lack of time as well as the different focus of this dissertation, these tests failed to appear.

### **9.2 FURTHER WORK**

The Grasshopper API does not satisfy the requirements completely for designing a standardised agent system for average users. Because of the object oriented approach of the Grasshopper API it is possible to expand it. The following expansions are conceivable:

- Generic easy database support.
- Generic (proxy-) access to these databases.
- Possibility to filter database contents.

- An hierarchically inter agency directory, sorted by topics like sale, information, products etc. that tells the agent what kind of information it can obtain.
- Some GUI-classes that provide the visualisation/change of the database content of an agent.

The principles how these expansions can be carried out, are described in the "Grasshopper Programmers Guide".

## 10 Abbreviations

API	<b>A</b> pplication <b>P</b> rogrammers <b>I</b> nterface
AWT	<b>A</b> bstract <b>W</b> indowing <b>T</b> oolkit
DAE	<b>D</b> istributed <b>A</b> gent <b>E</b> nvironment
GUI	<b>G</b> raphical <b>U</b> ser <b>I</b> nterface
IP	<b>I</b> nternet <b>P</b> rotocol
JFC	<b>J</b> ava <b>F</b> oundation <b>C</b> lasses
MVC	<b>M</b> odel– <b>V</b> iew– <b>C</b> ontroller
TCP/IP	<b>T</b> ransmission <b>C</b> ontrol <b>P</b> rotocol
WWW	<b>W</b> orld <b>W</b> ide <b>W</b> eb

## 11 Glossary

Agency	A runtime environment for agents.
Agent	Independent program that is able to migrate in a network.
Forte	Graphical design tools for Java applications.
Grasshopper	An agent system from the company IKV++.
Heavyweight Component	A system–dependent graphical component.
Java	Object oriented programming language from SUN.
Lightweight Component	A system–independent graphical component.
Place	An agency is subdivided by places.
Region	All agencies belonging to the same Region Registry are called a Region.
Region Registry	Central point of an agent system where all agencies and agents have to register.
Swing	Java API for graphical interfaces.
Virtual Machine	Emulation of a Java “Bytecode” machine.
Yahoo	Internet search engine.



## 12 Bibliography

### *Internet Resources:*

**Grasshopper Programmers Guide and Users Guide from IKV++**

<http://www.grasshopper.de/>  
<http://www.ikv.de/products/index.html>

**Java/Swing/Forte Documentation**

<http://java.sun.com>

### *Books:*

#### **Essential Guide to User Interface Design**

Wilbert o. Galitz  
Wiley Computer Publishing  
ISBN 0-471-15755-4

#### **Advanced Techniques for Java Developers**

Daniel J. Berg, J. Steven Fritzinger  
Wiley Computer Publishing  
ISBN 0-471-18208-7

#### **Thinking in Java**

Bruce Eckel  
Prentice Hall PTR  
ISBN 0-13-659723-8

#### **Mobile Agents**

William R. Cockayne  
Michael Zyda  
Manning Publications  
ISBN 0-13-858242-4

### *Magazines:*

#### **IEEE Communications Magazine**




July 1998

#### **Die Ergonomen kommen**

c't 1999, Issue 25

## 13 Management of the project

### 13.1 THE LEGEND OF THE TIME PLAN

	Marked calendar weak.
	Milestone, at milestones a chapter including the documentation is finished.
	Reserved time for compensation of delays.

### 13.2 COMPARISON OF THE TIME PLAN WITH THE REAL USED TIME

The time plan of the interim report (Figure 13–1) could not be met completely. The main delay has happened during the "Theoretical investigations". The Grasshopper system was more complex than expected. Also to understand the process of good interface design as well as finding literature that explains the process of designing interfaces, and not only the technical use of components, was very time consuming. Altogether there is an delay three weeks (Figure 13–2).

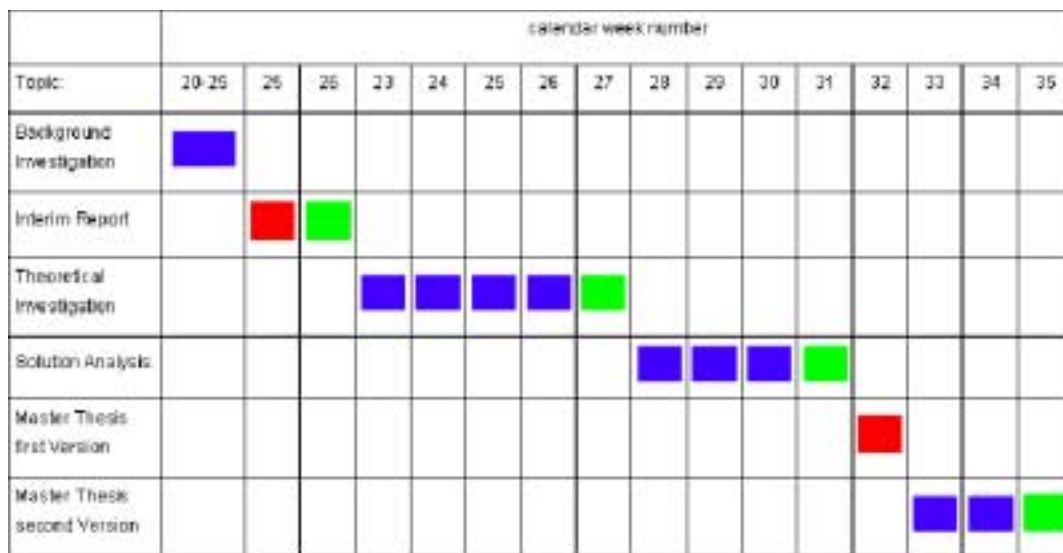


Figure 13–1 the time plan of the interim report



























































































	calendar week number														
Topic	20-25	25	26	27	28	29	30	31	32	33	34	35	36	37	38
Background- Investigation															
Interim Report															
Theoretical- Investigation															
Solution Analysis															
Master Thesis first Version															
Master Thesis second Version															

Figure 13-2 The time really needed