MSc in Distributed Computing Systems Engineering

Department of Electronic & Computer Engineering

Design and Implementation of a Flight Booking System on the Internet supported by Java-based Mobile Agents

Axel S Hallwachs

March 2001

MSc in Distributed Computing Systems Engineering

Department of Electronic & Computer Engineering

Design and Implementation of a Flight Booking System on the Internet supported by Java-based Mobile Agents

Axel S Hallwachs

Supervisor: Dr M J D Wilson

March 2001

Acknowledgements

The project is intended to develop an understanding and expertise within the broad topic areas covered by the MSc course Distributed Computer Systems.

The aim of the dissertation is to demonstrate the ability to carry out a project and to establish an in-depth understanding of a particular subject area. The first section begins with the background and the topic ‘Mobile Agents’. The section with a comparison of agent systems follows. After that the requirements, analysis, design and implementation of the Flight Booking System is described on the basis of a software lifecycle. The Interim Report is bound at the back.

The results of this dissertation were only possible, because of the great support and supervision of my supervisor Dr. Manissa Wilson. Her very helpful corrections and advises of my delivered draft documents contributed to instruct me to find the right way. I would like to thank Mrs. Dr. Wilson for her time intensive handwritten corrections of my documents and her e-mail communication.

This dissertation has occupied plenty of time. I would like to say ‘thank you’ to all the people who supported me with their considerateness in order to bring it to a successful end.

Esslingen, March 2001
Axel S Hallwachs

Abstract

General

Mobile agents are programmes that are able to migrate from host to host in a network, at times and places of their own choosing. The term 'software agent' has a spectrum of definitions. At one end of the scale are relatively simple, client-based software applications that can assist users in performing tasks. At the other end of the scale is the concept of software entities possessing artificial intelligence that autonomously travel through a network environment and make complex decisions on a user's behalf. The advantage of mobile agents is the reduction of network traffic and asynchronous interaction.

Theme of the Master Project

The project starts with a discussion about the need of mobile agents. Where are the differences to traditional solutions or, which possibilities do we have to solve the same tasks? An agent system has to run on every host, on which an agent should migrate to. The first part of the master thesis would be a comparison of different agent systems. The main topics that have to be answered are performance, security, portability, standardization, communication and resource management.

Mobile agents differ from applets, which are programmes downloaded as the result of a user action and that are executed from the beginning to the end on one user’s host. The second part of the master thesis shows an implementation of an agent application using one agent system discussed in the first part. This example could be a purchase agent travelling through the network and collecting special information, which could be compared to decide special actions.

Table of Contents

1 Introduction

The Internet is more and more involved in our daily life. It is as important as the telephone. The most relevant parts are the World Wide Web, e-mail, newsgroups and file transfer. The World Wide Web provides a graphical navigation through millions and millions of computers with web servers running on it. From a technical point of view, we can say that the Internet is a huge and open network of connected computers of different hardware, software, operating systems, etc. which can be accessed by everyone. But if we think of the Internet with an economical and commercial prospect it is much more than this. Global communication takes us to a global village in which we can trade with everyone.

More and more activities can be dealt with the help of the latest web technologies. E-commerce is a great topic to manage things like ordering, shopping, online banking, stock trading, auctions, software downloading, message sending and infinitely more things. The Internet offers nowadays much more than just ‘ trans-mission of information’ like in the earlier days. To access all the information you like is very useful and enjoyable. All people have the same access to the same sources independently of where they are. This is great – but the Internet is much more powerful.

There is an economical substance in the Internet. Conceivable possibilities are initiations like online orders practised by bookshops or music stores. The customer can choose products of every vendor independently of time and date all around the world. The sellers can offer their products to a much greater audience. E-commerce opens the efficiency of supply chains and can reduce costs. It changes the relationship of businesses and customers. Companies’ organisations will be changed because of competition advantages. The electronic exchange influences every enterprise, the logistic, the support and the purchasing department. You can develop a supply channel, which is never known.

If you provide goods, information or services with a secure Internet technology you talk about E-commerce. These are transactions to build a whole workflow on the Internet. Price arrangements can be created much faster as a reaction on the supply and demand than on traditional ways. The solutions are supply oriented and demand controlled.

Finance applications and enterprise resource planning are main topics. An electronic data interchange system manages the automatic obtaining of new material or a dynamic trade can be established from the raw material to the end user. [Sun00]

1.1 The Topic

Design and Implementation of a Flight Booking System on the Internet supported by Java-based Mobile Agents

The topic meets the necessary attributes of a project. Because of the distribution of the software over many computers we have a 'distributed systems architecture' and because of the software engineering aspects it is clearly relevant to the subjects of the course 'Distributed Computing Systems Engineering'. It is an area that has a great potential in the future. It's developments get more and more important because of the flexibility and mobility.

1.2 Description

Nowadays a lot of orders are dealt via the Internet. Everyone who wants to obtain information about offered flights has to check all travel agencies independently one after another. You move from one web page to the next and enter the same information with every single offer. If anybody doesn't know where to look, he or she has to search for all the travel offices first, to compare the prices for a special flight. Perhaps he or she doesn't know how to find the offices or where to obtain all the different services or offerings.

In conjunction with booking a flight, you often have to book a hotel room in the town you travel to. To get a free hotel room, you do the same actions as you did for booking a flight. That means, that you look for offers providing hotel rooms and you search the web in a time intensive action to compare the search results to get a cheap and nice opportunity. But you don't know if there is one you didn't recognize.

This is a great possibility for mobile agents, because you can construct a logic, which can be used again and again. Another reason for using mobile agents is the fact that the previous outlined scenario could be inversed. The customer would not have to search for the offers, but the sellers would have to provide their services.

One disadvantage of the description above is, that the user has to search explicitly and in detail for all offers he or she can find and it is always conceivable that he or she misses some opportunities.

In the approach of this project, the sellers carry out the first action and offer their products. They do this in a computer network. One part of the network is located

on the sellers' computers and one central location has a registry with all offered services combined with the vendors.

The user who wants to book a flight or a hotel room just has to start a mobile agent with the necessary information of the date and the cities and the agent can collect all resources, which are reachable. The user can rely on the result that all available offers are considered.

1.3 Why Mobile Agents?

The topic of all distributed object technologies is a synchronous message-passing paradigm whereby all objects are distributed but stationary. This paradigm is incomplete and needs to be enhanced in some fashion with additional paradigms such as asynchronous message passing, object mobility, and active objects.

Mobile agents provide a single uniform paradigm for distributed object computing, including synchrony and asynchrony, message-passing and object-passing, stationary objects and mobile objects.

Along with mobility, agents have the following unique, fundamental and important computational characteristics:

Object passing: When a mobile agent moves, the whole object is passed; that is, its code, data, execution state, and travel itinerary are passed together.

Autonomous: The mobile agent contains sufficient information to decide what to do, where to go, and when to go.

Asynchronous: The mobile agent has its thread of execution and can execute asynchronously.

Local interaction: The mobile agent interacts with other mobile agents or stationary objects locally. If needed, it can dispatch messenger agents or surrogate agents, which are all mobile agents, to facilitate remote interaction.

Disconnected operation: The mobile agent can perform its tasks whether the network connection is open or closed. If the network connection is closed and it needs to move, it can wait until the connection is reopened.

Parallel execution: More than one mobile agent can be dispatched to different sites to perform tasks in parallel.

2 A Survey of Mobile Agent Systems

Intelligent and frequently autonomous and mobile computer code known as agents represent the next great wave of innovation and development across the information sphere comprised of the Internet, Intranets, Extranets, World Wide Web, and countless other networked computer systems. This arena has increasingly become very active, rapidly evolving, and expanding in scope and importance. This technique can only be used, if an agent system is installed on every involved host as a middleware. Therefore a survey of available mobile agent systems is useful to find the system to employ for the project.

Most agent systems use Java as the development language. It would seem that the general acceptance of Java as the de facto Internet programming language gives rise to accelerated research in mobile agent technology. The survey comprises these systems into consideration, which allow programming the agents with the programming language 'Java'. This is the first criterion that has to be fulfilled in that survey.

2.1 Aglets

Agents are Java objects on the move. IBM Tokyo Labs has implemented Java objects that can move from one host on the Internet to another. Such objects can execute on one host, suddenly halt execution, dispatch to a remote host, and resume execution there. When the aglet moves, it takes along its program code as well as its state (data). A built-in security mechanism makes it safe for a computer to host only trusted aglets.

The Aglets' Workbench[1] is a first-of-its-kind visual environment for building network-based applications that use mobile agents to search for, access, and manage corporate data and other information. This visual builder is called 'Tahiti'.

Mobile network agents are programmes that can be dispatched from one computer and transported to a remote computer for execution. Arriving at the remote computer, they present their credentials and obtain access to local services and data. The remote computer may also serve as a broker by bringing together agents with similar interests and compatible goals, thus providing a meeting place at which agents can interact.

[1] section 2.1.3 Usage

The Aglets' Workbench makes it easier than ever to create mobile platform-independent agents based on the Java programming language. Its visual builder allows to compose personalized agents quickly that can roam the Internet, and the rich set of software components in this workbench enables agents to access corporate databases, search, travel, and communicate in a standardized and secure manner.

The agent system 'Aglets' models the mobile agent to closely follow the applet model of Java. It is a simple framework where the programmer overrides predefined methods to add desired functionality. [Lan96] defines an Aglet as a mobile Java object that visits Agent-enabled hosts in a computer network. An Aglet runs in its own thread of execution after arriving at the host. It is also reactive because it responds to incoming messages. The complete Aglet object model includes additional abstractions such as context, proxy, message, itinerary, and identifier. These abstractions provide 'Aglets' the environment in which it can carry out its tasks.

2.1.1 The Agent Transfer Protocol

The Agent Transfer Protocol (ATP) is used to transfer agents over the network. Aimed at the Internet and using Universal Resource Locators (URL) for agent resource location, ATP offers a uniform and platform-independent protocol for transferring agents between networked computers. While mobile agents may be written in many different languages and for a variety of vendor-specific agent systems, ATP offers the opportunity to handle agent mobility in a general and uniform way.

For example, any agent host will have a single and unique name independent of the set of vendor-specific agent systems it supports. ATP is implemented as an independent and fully documented package in the 'Aglets' Framework. Entirely written in Java, this highly portable set of classes provides a standard API for creating ATP daemons, connecting to ATP sites, and generating ATP requests and responses. This package is independent of any particular agent implementation.

2.1.2 Security

Security is of paramount importance to users of mobile agents. Receiving unknown agents from across the network is potentially an open invitation to all sorts of problems.

The ‘Aglets’ framework supports an extensible-layered security model. The first layer of security comes from the Java language system itself. Imported code fragments in agents are subjected to a series of checks, starting with tests to ensure that the code format is correct, and ending with a series of consistency checks by the Java bytecode verifier.

The next layer is the security manager, which allows users to implement their own protection mechanisms. Tahiti (section 2.1.3) implements a configurable security manager that provides a fairly high degree of security for the hosting computer system and its owner. The default security configuration is very restrictive. Any attempt of an agent to execute a file access that has not been granted will be regarded as a security violation, and the agent will not be allowed to perform that specific access.

The Java Security API is part of the third and final layer. It is a framework that makes it easier for agent developers to include security functionality in their agents. This functionality includes cryptography, with digital signatures, encryption, and authentication.

2.1.3 Usage

Tahiti is a visual agent manager based on the ‘Aglets’ Framework. Tahiti uses a GUI to monitor and control aglets executing on the computer. It is more than a system administration tool; it is a desktop tool for agent users in the same way that a web browser is the fundamental tool for Internet users.

Tahiti is an application programme that runs as an agent server. Multiple servers can run on a single computer by assigning them different port numbers. Tahiti provides a user interface for monitoring, creating, dispatching, and disposing of agents and for setting the agent access privileges for the agent server.

The following example shows the agent ‘HelloAglet’ that takes a string to another server. There are two servers on the host with the name ‘London’ and the IP address 210.11.12.13. One is started on port 434 and the other on port 7020. The agent starts on the server with port 434, migrates to the second server, prints out the string ‘Brunel University’, and returns to the initiating server with the sentence ‘I’m back.’. The demonstration on the next page shows the Tahiti windows for the phases: agent start, arrival at the destination server and the return.

First Phase

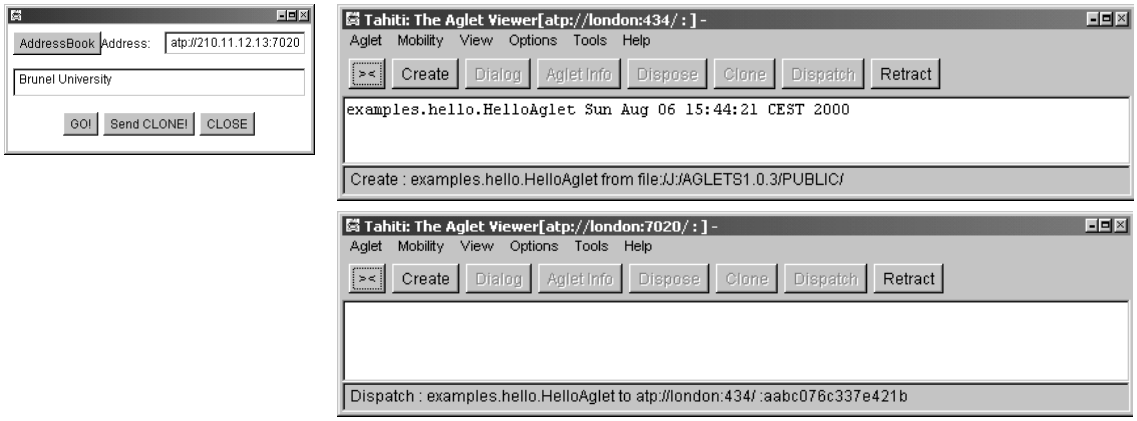


figure 1 first phase of the 'Aglets' session

A note with the name, date, and time indicates every agent on a server. All the agents are displayed in the list box in Tahiti. The agent 'examples.hello.HelloAglet' starts a popup window with the address to travel to and the string to show on the destination location.

Second Phase

The agent moves from one server to the other. There is no agent on the server with the port 434 anymore. The agent moves to port 7020 and shows the string.

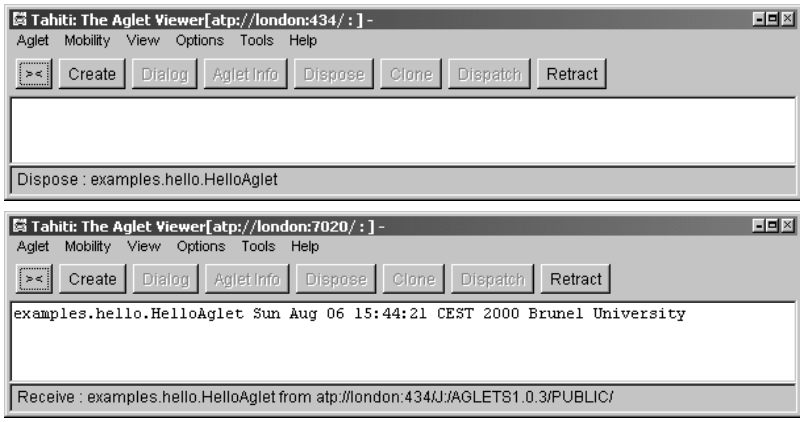


figure 2 second phase of the 'Aglets' session

Third Phase

The agent is back on the initiating server and has ended its route.

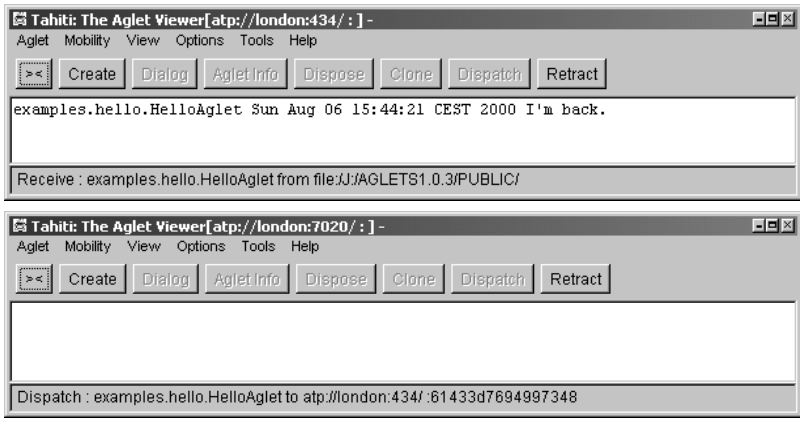


figure 3 third phase of the 'Aglets' session

The installation of the 'Aglets' system is more complicated than the installation of the other systems of this survey. There isn't an installation programme. The entire system variables have to be set individually and without any user's manual. It takes much more time to get it started than starting the other agent systems. One reason is, that it is based on the java development kit (JDK) 1.1, which is a two-year-old technology.

The development of the 'Aglets' agent system is stagnating. There have not been any further releases for the last two years. This is a very long time in the rapid development periods of the information technology. The 'Aglets' system doesn't support the latest Java version JDK 1.3, it is still based on JDK 1.1. The new features between these two development kits especially in graphical user interface design and in using the collection classes are so enormous, that it is important to employ the newer version 1.3.

2.1.4 Implementation

The abstract class `Aglet` defines the fundamental methods of a mobile agent, for example `dispatch()`, to control its mobility and lifecycle. All mobile agents defined in 'Aglets' have to extend this abstract class. The `dispatch()` method causes an aglet to move from the local host to the destination given as the argument.

The `AgletProxy` interface acts as a handle of an agent and provides a common way of accessing the aglet behind it. Since an aglet class has several public

methods that should not be accessed directly from other aglets for security reasons, any aglet that wants to communicate with other aglets has to obtain the proxy object first, and then interact through this interface. In other words, the aglet proxy acts as a shield object that protects an agent from malicious agents. When invoked, the proxy object consults the `SecurityManager` to determine whether the current execution context is permitted to perform the method. Another important role of the `AgletProxy` interface is to provide the aglet with location transparency. If the actual aglet resides at a remote host, it forwards the requests to the remote host and returns the result to the local host.

'Aglets' objects communicate by exchanging objects of the type `Message`. A message object can have a string object to specify a kind of message and arbitrary objects as arguments. The following methods of the `Aglet` class are supposed to be overridden by a subclass to allow an aglet to implement its own specific behaviour:

The method `onCreation(Object init)` is supposed to be called only once during its lifecycle, when it is created. 'Aglets' programmers have to use this method to initialise an aglet object, because the Aglet API (`dispatch(URL)` for example) is not available in the constructor. `OnDisposing()` is called when an aglet is disposed. The aglet is supposed to release any resources previously allocated. It can perform additional actions in response to its own disposal.

The `run()` method in the `Aglet` class is called whenever an instance is reconstructed, that is, when the instance is created, when it is cloned, when it arrives at the destination, and when it is activated. Because this method is called whenever it occupies the context, this is a good place to define the common task.

All messages sent to the aglet are passed to the `handleMessage()` method. Aglet programmers can check whether the incoming message is a known message, and can perform the task according to the kind of message.

Serializing an Aglet Object, a Non-Proxy or a Proxy Objects

If an aglet is dispatched, cloned or deactivated, it is marshalled into a bit-stream and later unmarshalled. The 'Aglets' system uses object serialization of Java to marshal and unmarshal the state of agents, and tries to marshal all the objects that are reachable from the aglet object.

All objects to be serialized have to implement either `java.io.Serializable` or `java.io.Externalizable`. If there is a non-serializable object that is directly or

indirectly referenced by an aglet, the reference should be declared as `transient` to indicate that it shouldn't be transmitted.

A non-proxy object, which is reachable from an aglet, is migrated by copy. This means that once serialized, an object shared by multiple aglets is copied and is no longer shared after the specific operations. When a proxy object is migrated, it keeps the aglet ID and its address, and restores the reference to the original aglet object. The `AgletProxy` object can keep the reference to the actual aglet even if the proxy is transferred to a remote host or deactivated, as long as the aglet resides in the same location.

2.2 Grasshopper

'Grasshopper', the agent development platform launched by IKV++ in August 1998, enables the user to create all kinds of applications based on agent technology. Because 'Grasshopper' allows software agents to move between different systems and to execute various tasks in the process, the platform is perfectly suited for distributed applications. It supports the java development kit 1.2 and the latest one 1.3.

2.2.1 The Topology

The 'Grasshopper' platform in separated into different parts. These are a Region, a Region Registry, an Agency, a Core Agency and a Place.

The Region concept assists the management of distributed components within a 'Grasshopper' environment. Agencies, Places and agents can be linked with a Region. That means that they enrol themselves at the Region Registry and are known in that Region. The Region Registry contains all information about an Agency connected to the Region. Every service of every new component can be used by all other registered components.

The Agency is the execution environment for stationary and mobile agents. Generally one Agency is located on one computer. If there is more than one Agency on one computer, they differ in the port number. Figure 4 shows a 'Grasshopper' Agency that includes the Core Agency and one or more Places. The Core Agency represents the minimal functionality an agent requires to be executed. There are communication, management, persistence, registration, security and transport services in an Agency. A Place is a logical group of agents within an Agency. The agents have a common context. That means that there is a

Place for these agents who want to trade or a Place for other agents who want to offer something. A Place should have a describable name.

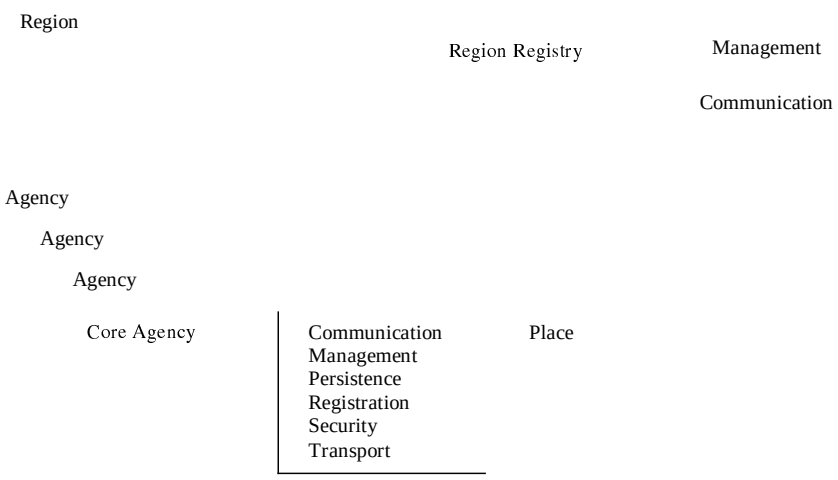


figure 4 topology of the 'Grasshopper' agent system

2.2.2 Mobility

Java Remote Method Invocation (RMI) enables the call of methods of objects located on foreign Java virtual machines. 'Grasshopper' solves RMI via a proxy object. That means that the communicating objects aren't linked directly.

Agent migration is a remote execution. That means, that the programme is transmitted to the place, where it is executed. It is possible to implement the method `move()` at any context of the source code. The parameter of this method is the URL combined with the protocol to use, the IP address of the destination host and the port number. If a migration is initiated, the agent thread is stopped and the data and the code are serialized. The Agency on the destination host starts a new instance of the agent and begins with the execution first and then announces the successful transmission to the previous host to delete the agent on its agency.

2.2.3 Communication

The communication concepts of the ‘Grasshopper’ platform are realized in the Core Agency communication service. This service provides the location independent and transparent access between agents and Agencies. Remote interactions are realized with IIOP, RMI or with sockets. Figure 5 shows the situation. To guarantee a secure transmission, the Secure Socket Layer Protocol can be used.

‘Grasshopper’ can dynamically choose the right protocol and the right port within a Region. For connections with objects outside the Region, the URL has to include the protocol, the IP, the port and the place name. The alternative to that communication service is using CORBA with the IIOP protocol. That approach works with the object implementation on a server and stubs for the clients.

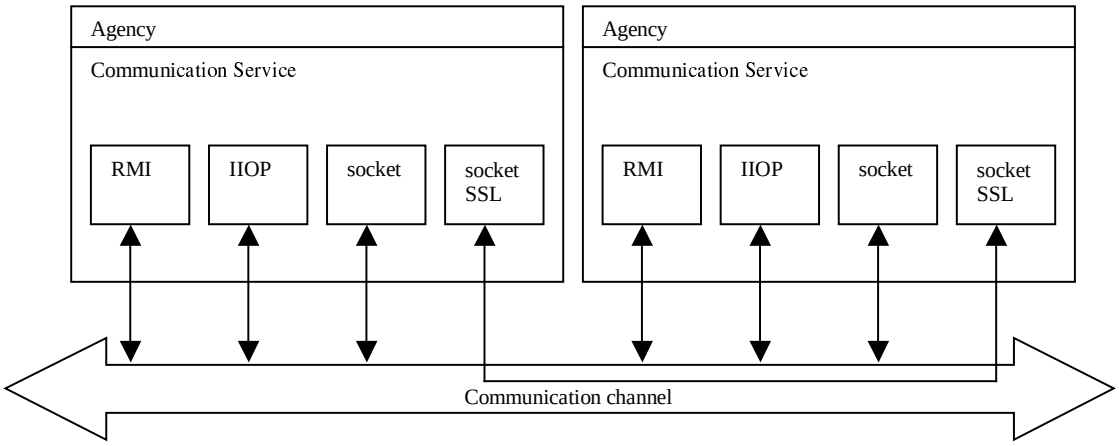


figure 5 ‘Grasshopper’ communication service

Stubs are client side representations of the server objects. Skeletons include the implementation of a method and are the client view for a server.

The communication service is used for the agent transport and for searching agents within one system. On the other hand it is possible to invoke methods of other agents running on other Agencies located on remote hosts. This is reached with proxy objects that are invoked. A proxy object moves the call to the server of the real object. The use of proxy objects facilitates internal commands of the communication.

2.2.4 Security

'Grasshopper' divides the security aspect into two parts. The external security protects all actions of the communication system. The X.509 and the Secure Socket Layer Protocol is used. The internal security protects the interfaces and the resources of the agent system against unauthorized use by an agent.

2.2.5 Usage

Figure 6 shows the properties of an agent system. The communication service offers adjustments for the different protocols like socket, IIOP or RMI and for the security. Another preference can be made for the desktop and the catalogue to manage the start of agents.

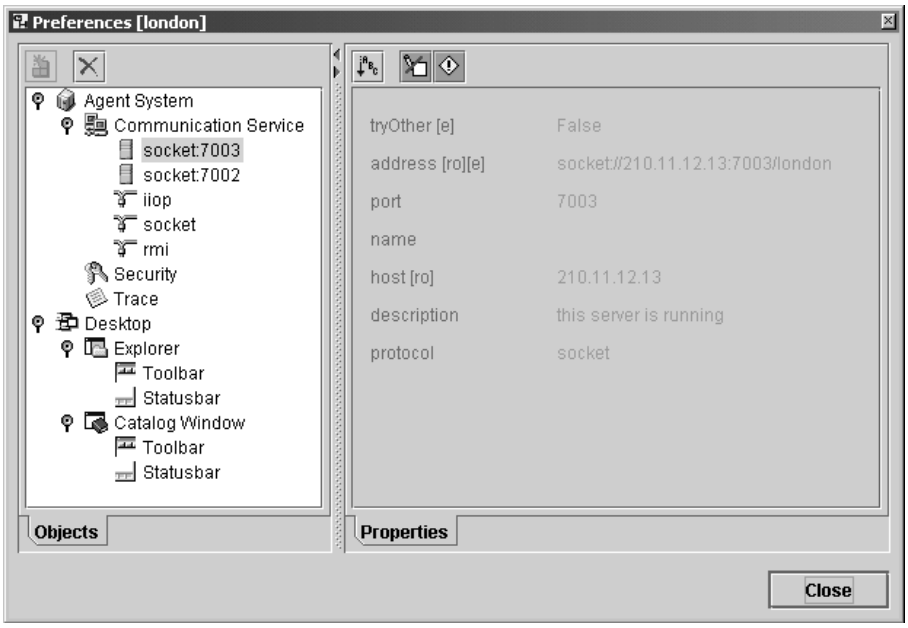


figure 6 'Grasshopper' agent system properties preferences

If an agent wants to know which Agencies, which Places or which services exist in a Region, it can request that information from the Region Registry. The first window of the example on the next page shows a registry. Stationary agents represent services. A mobile agent can use a service, after it has obtained the location where to find it from the registry and whether the stationary agent is alive or not.

The example on the following page shows the 'BoomerangAgent', which moves to another Agency specified by the IP address and the port. There are two Agencies registered at the registry. One with the name 'first' and another called 'second'.

Situation One

The Agency ‘first’ locates the Boomerang Agent. The agent has an input dialog to enter the destination address of Agency ‘second’ to define where to go.

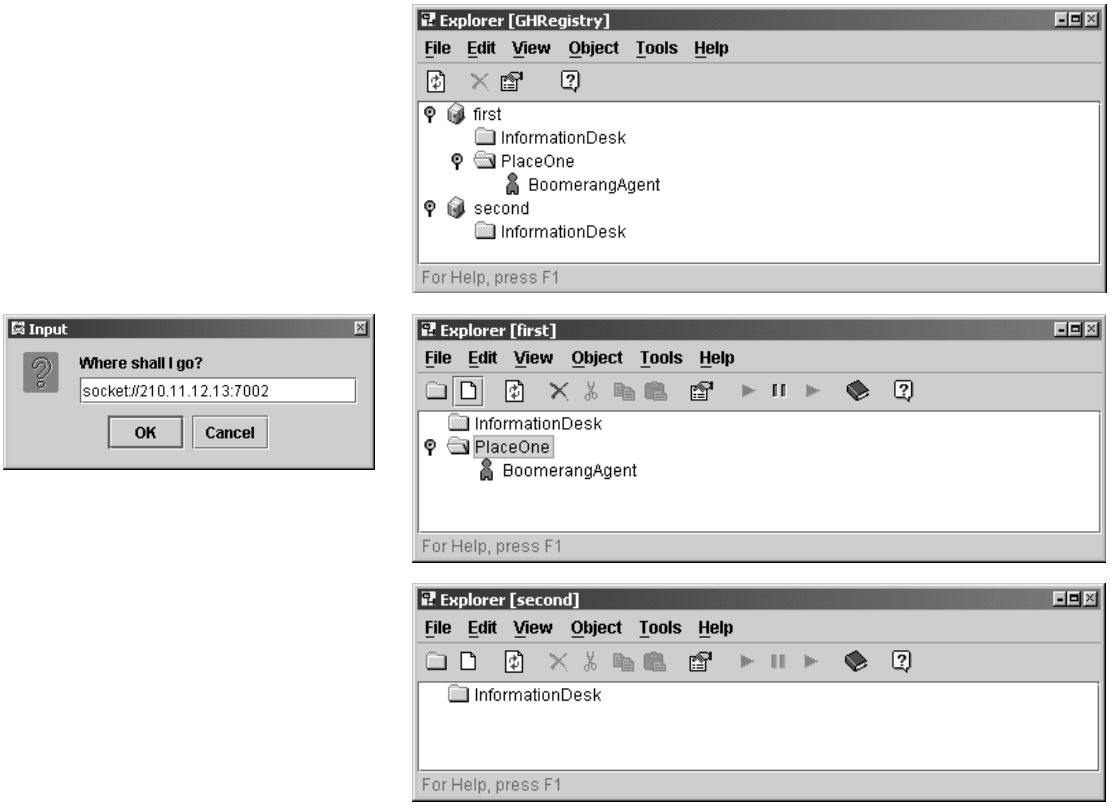


figure 7 phase one of a ‘Grasshopper’ session

Situation Two

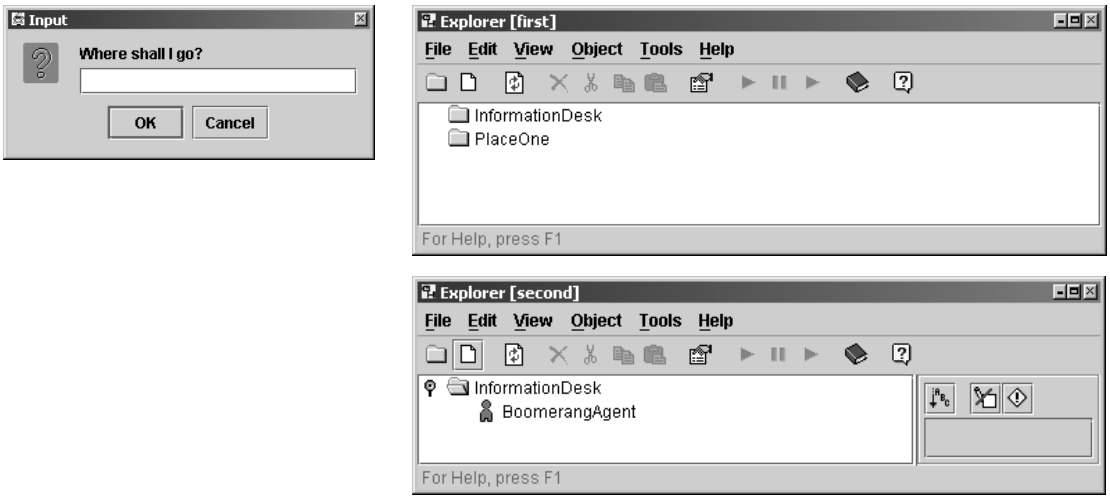


figure 8 phase two of a ‘Grasshopper’ session

The agent migrates from Agency ‘first’ to Agency ‘second’ after the user entered the address in the popup window. It is removed from Agency ‘first’ and stays on Agency ‘second’ now. This movement can be repeated again and again.

2.2.6 Implementation

The functionality of ‘Grasshopper’ is provided on one hand by the platform itself, i.e. by Core Agencies and Region Registries, and on the other hand by agents that are running within the Agencies, in this way enhancing the platform with new capabilities. The article [ikvPG98] mentions that the following possibilities regarding the access of ‘Grasshopper’ functionality must be distinguished:

Agents can access the functionality of the local Agency by invoking the methods of their super classes `StationaryAgent` and `MobileAgent`. These super classes have the super class `Service` and are provided by the platform in order to build the bridge between individual agents and Agencies, and each agent has to be derived from one of these classes.

Agents as well as other distributed agent environment components, such as user applications, are able to access the functionality of remote Agencies and Region Registries. For this purpose, each Agency offers an external interface, which can be accessed via the communication service.

A ‘Grasshopper’ agent consists of one or more Java classes. One type of these classes builds the actual core of the agent and is referred to as the agent class. This class has to implement the method `live()` which specifies the actual task of the agent. The complete task of a ‘Grasshopper’ agent to be performed during its entire life time, has to be covered by the method `live()`. The statements comprised by the method `live()` are separated into several execution blocks. Each execution block is completely executed within a single place, and the last method of each block is the `move()` method.

Every agent has a state that changes after every migration. To retrieve information about an actual state a proxy object of the agent is used. The method `action()` should be overridden to implement individual tasks, because it is called after invocating the agent with the `invokeAgentAction()` method. The method `init()` is automatically called only once during the whole life time of an agent with the first creation. In contrast to the `init()` method, an agent’s constructor is invoked after each migration. After creating a Region Registry proxy, all methods provided by the Region Registry may be performed remotely by the client object. The method `lookupServices()` offers the access to the service info.

2.3 Voyager

'Voyager' is the first 100% Java agent-enhanced Object Request Broker (ORB) developed by the company ObjectSpace. It combines the power of mobile autonomous agents and RMI (remote method invocation) with complete CORBA support and comes completely with distributed services such as directory, persistence, and publish subscribe multicast.

Proclaimed as an agent enhanced object request broker in Java, 'Voyager' offers several advanced mechanisms that could be used to implement agents. The 'Voyager' agent model is also based on the concept of a collection of Java objects. It has integrated native CORBA support, and can communicate with other CORBA systems regardless of their implementation language.

2.3.1 The Topology

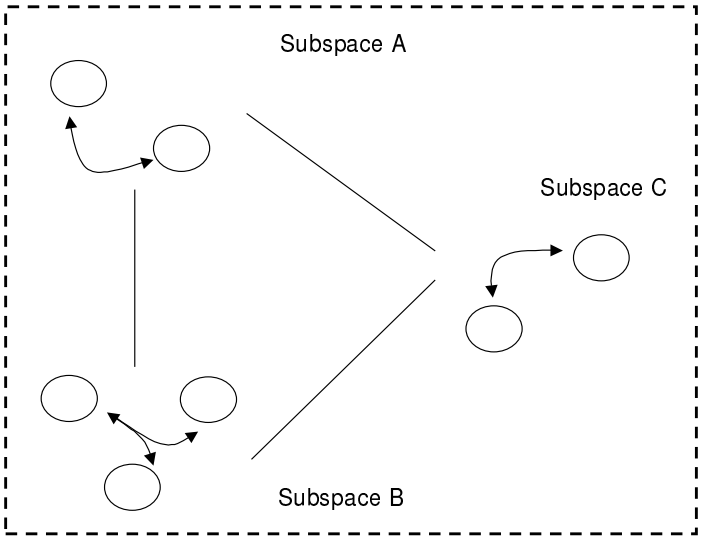


figure 9 topology of the 'Voyager' agent system

The figure above shows the structure of a 'Voyager' system. The dashed line is the border of all involved participants and it is called the Space. A subspace includes some agents on one or more computers. A Space contains distributed objects on different computers and can span multiple programmes. Linking together one or more subspaces creates a Space. A subspace is a local container of objects. A message or event sent via a proxy into a subspace from another subspace is delivered to every object in the local subspace. The more inter-connected the subspaces are, the more fault-tolerant they become in the face of individual network failures [Obj98].

2.3.2 Security

A security manager can be installed in a 'Voyager' programme. Once a security manager is installed, it is active for the duration of the programme. It cannot be uninstalled or replaced. Each time an object attempts to execute an operation that could affect security, the Java run-time machinery checks with the programme's security manager to determine whether the operation is permitted. If the application has no security manager, or if the security manager permits the operation, 'Voyager' proceeds as normal. If the operation is refused, a run-time security exception is thrown.

2.3.3 Services

Naming

A naming service allows names to be associated with an object for later lookup. There are many different implementations of naming services, including:

- 'Voyager' federated directory service
- CORBA naming service
- JNDI, Java Naming and Directory Interface
- Microsoft Active Directory
- RMI registry

When a naming service is used to bind a name to an object, it adds a unique prefix to determine later the type of the naming service directly from the name. The 'Voyager' `Namespace` class takes advantage of these prefix codes to provide a single, simple interface that unifies access to one or more of these naming services. New naming services can be dynamically plugged into `Namespace`. Each of the following static methods uses the name's prefix to determine which underlying naming service to access. Method `lookup(String name)` returns a proxy to the object associated with the specified name, or null if no such object is found. The method `bind(String name, Object object)` associates the specified name with the object, or throws an exception if the name already has an association. To associate the specified name with the object, and replacing any previous association if present, the method `rebind(String name, Object object)` is used, and `unbind(String name)` is for disassociating the specified name.

Persistence

'Voyager' does not include any kind of persistent storage, since this is best left to database products. Facilities like the activation framework that saves and restores objects need to access the complete state of an object.

2.3.4 Utilities

The **voyager** utility starts a 'Voyager' server from the command line. One valid argument is the port on which to start the server and the application.

The **igen** utility creates a default interface from a class. The interface has all the public methods of the original class, and its name is like the original class name prefixed with an 'I'. To use the igen utility, the class name is specified without the .class or .java extension. Igen searches the directories, .zip files, and .jar files in the classpath for the specified file and generates an interface for the class.

The **cgen** utility can translate interface definition language (IDL) files to and from Java. To translate an IDL file to Java, cgen is executed with a list of the IDL files. The file names should end with an .idl extension. To translate a Java file into IDL, cgen is executed with a list of .java and/or .class files, omitting any extension.

2.3.5 Implementation

The class library has an `Agent` class that developers can subclass to implement 'Voyager'-style agents. The 'Voyager' agent is designed to take advantage of the 'Voyager' ORB features, which make extensive use of Java's reflection mechanism. A 'Voyager' agent can communicate by calling methods or using 'Voyager' Space technology, a group event and message multicast facility that ObjectSpace claims to be more scalable than simple communication mechanisms like in 'Aglets'.

A message sent via a proxy is executed according to the following rules:

- If the destination object is in the same programme, the message is delivered just like a regular Java message. The arguments are not serialized or copied, resulting in very high performance.

- If the destination object is in a different application, the arguments and return values must be sent across the network.

Sometimes an object needs to know that it is about to move or has just been moved. For example, a persistent mobile object may need to remove itself from the origin's persistent store and add itself to the destination's persistent store. 'Voyager' provides this capability through the `IMobile` interface. If an object or any of its parts implements the `IMobile` interface, then it will receive callbacks during a move in the following order:

The most important methods for migration are:

`postArrival()`
At this point, the copy of the object has become the real object, and the move is deemed successful and cannot be aborted. `PostArrival()` is executed on the copy of the object at the destination immediately prior to the user-supplied callback, and is typically defined to perform activities such as adding the new object into persistent storage.

`postDeparture()`
This method is executed on the original object at the source, and is typically defined to perform activities such as removing the old object from persistence. Messages sent to the stale object via a proxy will be redirected to the new object, so `postDeparture()` should not utilize proxies to the original object.

To make an object become a mobile autonomous agent, the method `of()` of the class `Agent` is invoked to obtain the object's agent facet, and then the methods defined in `IAgent` are used:

The method `moveTo()` moves the agent to the programme with the specified URL and restarts then by executing a one-way callback with optional arguments. With `getHome()` the agent returns to the home, which is the URL of the agent location when its agent facet was first accessed.

A successful call to `moveTo()` conceptually causes the thread of control to stop in the agent before it moves and to resume from the callback method in the agent after it has moved. Therefore, only exception-handling code should follow a `moveTo()` method.

2.4 Comparison Result

All agent systems consist of Java class libraries. These are used to develop the mobile code with the special classes.

An agent application is considered as a special application that requires two parts: the mobile part and a host part that resides on a computing device at a network node. There is also a service point or location concept that serves as a mediator between the mobile agent and the services offered. All compared agent systems have a repository or registry to administer the enrolled agents.

Current agent systems assume that the operating environment is heterogeneous, so the first consideration in the choice of such systems is how to deal with platform independency. Java offers that feature. It runs on every platform for which a virtual machine is provided.

Another absolutely crucial issue for a mobile agent system is how to guarantee certain security levels so that the agent is protected from the host, hosts are protected from the agent, agents are protected from each other, and hosts are protected from each other. All systems have their own security technology and provide a secure transmission.

Agent mobility mechanisms include remote invocation (also known as remote execution), cloning, programming language support, middleware, and code on demand. Most systems discussed use application protocols on top of TCP/IP for transport of agent codes and states. Systems based on Java make use of the programming language features, such as support of a kind of RPC called remote method invocation RMI, object serialization, and reflection. Often the agent states and codes are transformed into an intermediate format to be transported and restarted at the other end.

A protocol is used to transmit the agent code and data. IBM has implemented a specific protocol for the ‘Aglets’ system. It communicates with ATP (aglets transfer protocol), an application-level protocol that works with a TCP socket. ‘Grasshopper’ supports the protocols plain sockets, Java RMI (remote method invocation) and IIOP (Internet inter-ORB protocol) of CORBA. ‘Voyager’ is designed as an ORB (object request broker) and handles objects on another host as remote objects. ‘Voyager’ uses Java object serialization and reflection extensively in its transport mechanism. The user of an agent system or of an agent application doesn’t want to be concerned with the communication protocol. It is transparent and the user doesn’t have to care how it works. ‘Grasshopper’ is the most flexible one, because the programmer has the possibility to choose between

three different protocols, which are all vendor-independent, and fundamental communication methods.

The ‘Voyager’ agent system offers a command line execution and agent control and hasn’t got a human computer interface. This is very convenient for a professional environment. But if the appearance and the movement of agents should be observed easily, a graphical user interface is very suitable to show the migration and the location of every agent. Especially for a project like this, the use of a visual representation of the agents gives a fast impression of what is going on and the commands for the command line are much more complicated than using a GUI to get the same result. This is the reason why the ‘Voyager’ agent system isn’t applied in this project.

Java is a fast growing and developing language. Its latest version is the development kit 1.3. This release has significant differences to its predecessors in the fields of designing graphical user interfaces and using the collection classes. Therefore it is important to use the latest version. The usage of the ‘Aglets’ system showed, that it just works with version 1.1 properly. Using version 1.1 means that you don’t have the great benefits of the Swing class library for programming GUIs, the collection classes and performance enhancements. There have not been any further progresses in the development of the ‘Aglets’ system during the last two years. This is the reason why the ‘Aglets’ agent system isn’t applied in this project.

The ‘Grasshopper’ agent system gives the possibility to choose between different transmission protocols. That is useful in an environment with different agent systems. It works with the Java version 1.3 and offers the latest features of Java. The handling, migrating and executing of agents is visualized with a graphical user interface. The Region Registry is a logical unit that can be used easily. The class library has extensive API documentation to find all classes and methods. Under those circumstances the decision is to take the ‘Grasshopper’ system. It has three advantages compared with the other systems.

3 Requirements of the Flight Booking System

The first step in achieving a successful software project is to analyse the client's current situation and to summarize the desires a customer wants as precisely as possible. The collection of requirements is useful for a client and a developer, because both see whether they talk about the same topic. The requirements analysis must establish the primary functions to be included. The aim is to define what is required, not how it can be achieved.

The specification document must be complete and detailed because it is virtually the sole source of information available in the design phase for drawing up the design. Every specification document incorporates constrains that the product has to satisfy.

3.1 Aims and Objectives

The aim of this project is to develop a flight booking system for an independent search for offered flights and hotel rooms.

The overall aims of the proposed project are:

- To provide a user the possibility to book a flight independently of a special travel agency or a special offer.
- To give all airlines the opportunity to offer their flights to every one who is interested in.
- To book a hotel room in conjunction with the flight. The hotel companies have special prices for the hotel rooms.
- To compare the search results of the different offers and give the user the possibility to choose one.
- To define resource protection against non-authorized access to the data of the agent.

3.2 Requirements Listing

The following requirements are numbered and structured in a top down order.

Requirement 100

The system should be accessible by everyone to use the services. Different providers offer their services without knowing of each other.

Requirement 120

The workflow that should be passed is information requesting, data collecting, data preparing and comparing, offer choosing and booking. In other words - the main topics are planning, selecting, getting price information, booking, paying and confirmation receiving.

Requirement 140

The Flight Booking System should be designed to find flights and hotel rooms with an easy request, independently of different providers. Flight and room offering and booking are the two services, which are offered.

Requirement 200

The user should enter his or her data via a human computer interface or a graphical user interface.

Requirement 220

First the user should decide if he or she wants to search for a single flight (outward flight) or an outward and return flight. The return flight starts at the destination of the outward flight and ends at the originating city of the outward flight. Then the user has to choose how many flights he or she wants to book, a departure and a destination city and the start and end date. The final action of the user is the choice of whether a hotel room at the destination city is requested or not.

Requirement 300

It is not mandatory to book a combination of an outward flight, a return flight and a hotel room. It should be possible to enter only one flight and no hotel reservation.

Requirement 400

It is not possible to book a hotel room without a flight. The first and main intention of the system is to book a flight and then to book a hotel room.

Requirement 500

The flight providers are airlines. They store their data in a file. The assumption for this project is that the flights take place weekly and that all seats in the aeroplane have the same price. Therefore one record of data consists of the originating and the destination city, the weekdays on which the flight takes place, the departure and the arrival times, the price and the seats available.

Requirement 600

The hotel providers are international hotel chains. They have hotels in some destination cities. The convention for this project is that they all offer single rooms for the same price. They store their data in a database. One record of data consists of the town in which a hotel is located, the number of available rooms during a specific time period and the price for a particular flight provider. The hotel chains offer their free rooms at a central host for all cities.

Requirement 620

The flight providers have partner hotels with which they have special contracts and special prices for the rooms. The convention for this project is that all hotel rooms are single rooms and every room at one location of a provider has the same price. The price only differs from the town and the flight provider that requests for it. Different flight providers could have the same hotel chain as a partner, but the room prices are different because of the contract and the quota a flight provider buys. If the flight provider were no partner of the hotel provider, the price for the rooms would be more expensive.

Requirement 700

If the system doesn't find a flight, the mobile agent won't search for a hotel room, because the user can't travel to the destination city. That means 'no flight' is the knockout-criterion for the application. A note tells the user that no flight was available.

Requirement 720

If the system finds a flight, the number of requested flights reduces the amount of

free seats for that flight. This is done with every flight provider, because the system doesn't know which one the user will take. After the user has chosen one flight or one flight and hotel combination, a confirmation is sent to that flight provider. After a specific time that means after a timeout all the other flight providers increase their amount of free seats again to the previous number of free seats.

Requirement 800

After the decision of the user, which offer to take, the credit card number of the user is sent with the confirmation to the flight and hotel provider to book the journey.

Requirement 900

The system should show the use and the benefits of mobile agents. It should be a demonstration application for this technique. The mobile agents should search and collect the data.

Requirement 1000

Stationary agents should cooperate with mobile agents and they should handle the requests with their services.

Requirement 1100

All stationary agents should be listed at a central place. This repository is the head host of the agent system and offers information about the distributed services that have enrolled themselves at that location. It should be possible for the mobile agents to retrieve all kinds of available services.

Requirement 1120

The mobile agent builds its itinerary with the information of the centralized agent system repository.

Requirement 1200

The development and runtime environment should be Java 2 with the java development kit 1.3.0. The agent system used should be 'Grasshopper', as stated in the comparison of different agent systems[2].

[2] section 2.4 Comparison Result

3.3 A typical Scenario

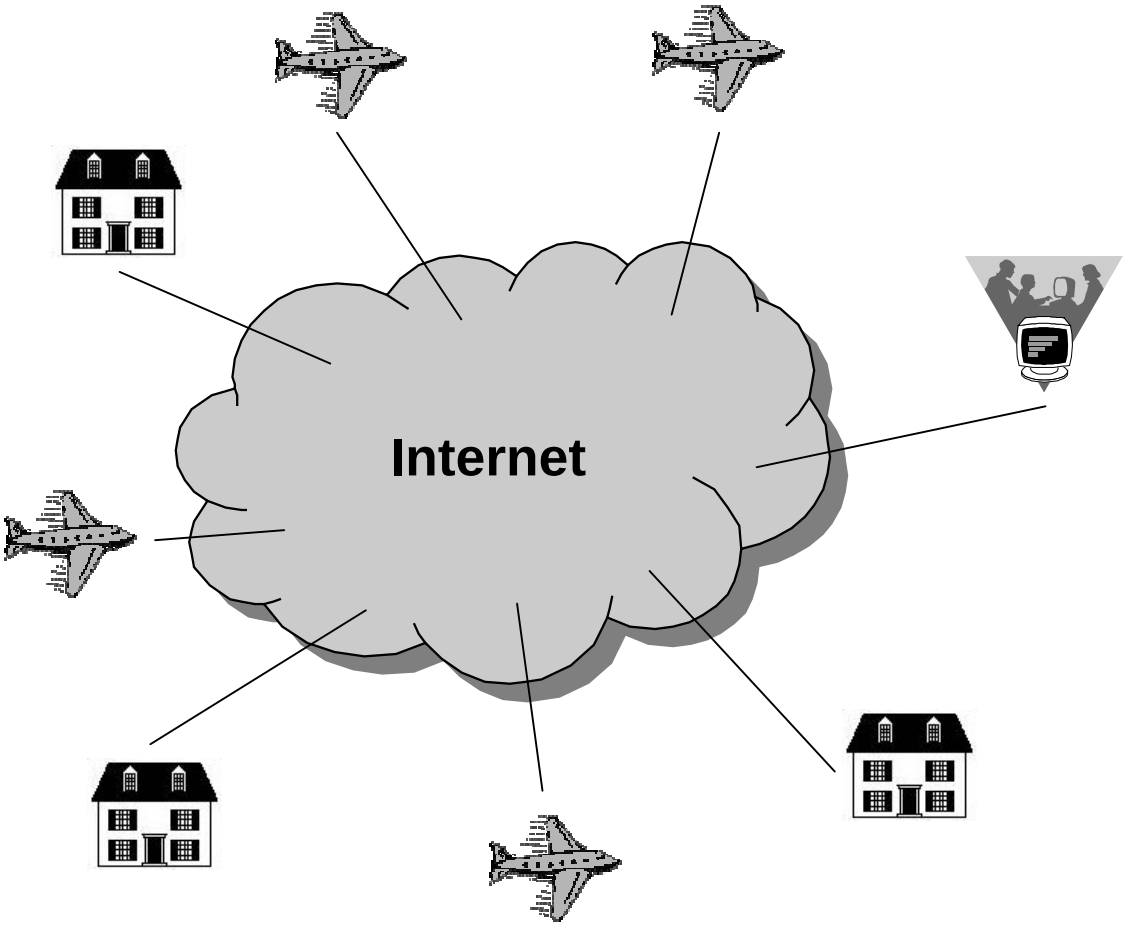


figure 10 symbolized project background

Figure 10 shows the flight providers, symbolized by airplanes, and the hotel providers, symbolized by houses. The terminal with people, on the right, is the user. It is the starting point on which the application is initiated. All symbols of the figure are hosts locating the latest information about what the provider offers. All these hosts consist of a computer with the agent system on it. Mobile agents can access the services of the providers, realized with stationary agents running on the hosts. If the user starts a search, the mobile agent requests the locations of the services from the agent system registry and migrates to the different flight providers. If the agent has found a flight, it activates another mobile agent to look for a free hotel room. The result will be taken to the point of departure and displayed to the user. With this technique, the user can rely on the information he or she gets, because this is the latest data from every provider. The collection is based on an independent comparison of all possible offers.

4 How to manage the Analysis Phase?

There are two ways of looking at every structured software product. One way is to consider just the data, including local and global variables or parameters. Another way of viewing a product is to consider the actions performed on the data, that is, the procedures and the functions. This division was the fundamental approach to the structured paradigm.

A data item cannot change unless an action is performed on it, and actions without associated data are equally meaningless. The object-oriented technique gives equal weight to data and actions and comprises both [Scha99]. All objects, which are found in this phase, include the data (more precisely, the attributes) and the methods to act on the data. A well-designed object has high cohesion and low coupling and models all aspects of one physical entity. That means, that an object uses information hiding and encapsulates all its attributes and composites with closely related methods and secondly just has few connections to other objects or a minimal set of well-defined channels. That implies low maintenance costs and ease of enhancement.

One huge advantage of object orientation is the equal comprehensibility of the context of the customer and the software developer. Both parties talk about the same objects, because everyone thinks of entities of the real world in the problem domain. The object model is an abstraction; a concentration on the essential part and it describes the possible behaviours.

Object-oriented analysis (OOA) is a semiformal specification technique for the object-oriented paradigm. OOA consists of three steps: use case modelling, class modelling and the dynamic modelling or object modelling [Hau97].

4.1 The Use Cases

The aim of the use case model is to determine how the product computes the various results. A use case model is the external view on a system. It explains what the application should perform. There is no sequence in using the use cases. A use case just describes one usage what the user can do with the system. An associated scenario shows the workflow of one use case. There could be another scenario for a different way or an error handling.

The first component of the model is a group of actors that interact with the system via triggering a use case. These actors aren't part of the system. The second component is the collection of use cases and their descriptions, the scenarios. A

use case describes the functionality of the product to be constructed and is concerned with the overall interaction between the system and the actors [Pro99].

The use case model in figure 11 shows six use cases. The user, for whom the application is developed for, interacts with three use cases: request a flight, request a hotel and choose order. The relationship between request a flight and display result is a uses-relationship. That means, that a user doesn't trigger it, instead it is used by two other use cases.

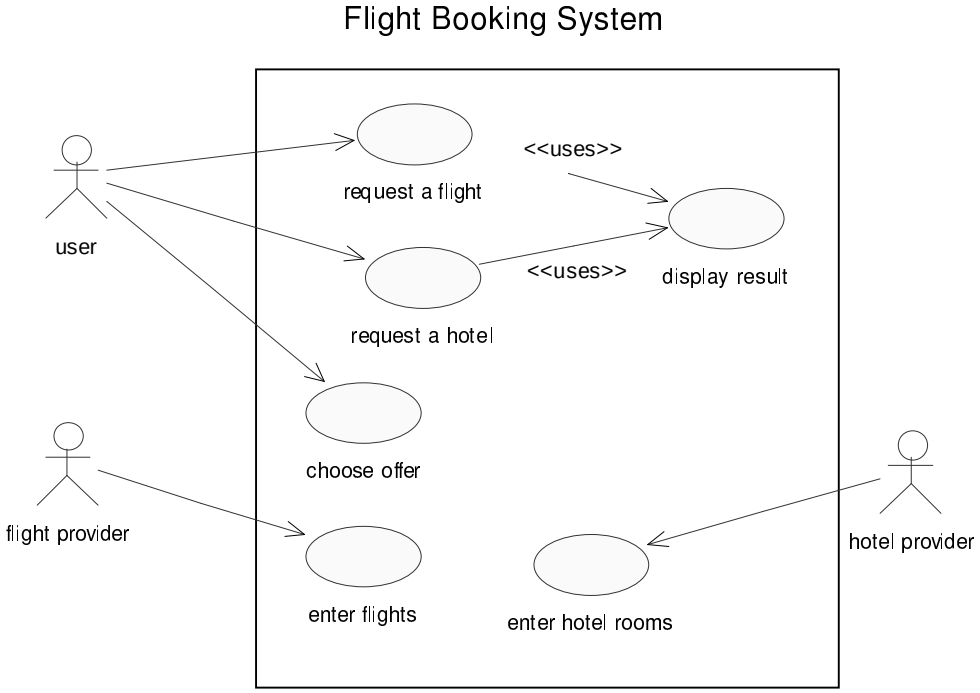


figure 11 use case model of the Flight Booking System

The actor 'user' requests a flight or a flight and a hotel room. After the result is displayed, the user chooses one offer. The two other use cases are maintenance tasks for the flight and hotel providers.

The scenarios are specific instances of the use cases. Sufficient scenarios should be studied to give the OOA team an extensive insight into the behaviour of the system being modelled. This information will be used in the next phase, class modelling.

A scenario is a textual description of a basic sequence of events with the description of the actor, involved actors and alternative flows. The requirements put up in the first period of the development cycle help to phrase the scenarios and to mention all necessary topics.

4.2 The Scenarios

Use Case 1: 'request a flight'

Actor: user

Sequence of events

1. The user starts the application with a **human computer interface** and enters the **flight data**, consisting of the departure day and the starting and destination airport.

2. The user enters the amount of flights.

3. The application collects all the **flight providers** at a **central repository**.

4. The application starts a **mobile agent** that moves from one flight provider to the next and contacts them via a **stationary agent**.

5. The stationary agent compares the available data with the entered information of the user.

6. If some **data match**, the stationary agent transfers a booking possibility to the mobile agent. The matching data contains the airline name, the amount of available places in the airplane and the start and landing time.

7. After the trip the agent returns to the **home destination** of the user.

Alternative Scenario

If the user wants additionally a return flight, he also has to enter the return day at the beginning of the request.

The input of the date is fault tolerant. If a wrong date is noticed, the system will tell the user about this mistake.

Use Case 2: 'request a hotel room'

Actor: user

Sequence of events

1. The user requests a flight like in scenario one. The request of a flight is mandatory to request a hotel room.

2. The user has a check box on the graphical user interface to indicate that he or she wants to order a hotel room.

3. All flight providers have contacts to some **hotel providers**. After the stationary agent of a flight provider has found a matching flight, **another mobile agent** starts to look for available hotel rooms.

4. The new started agent has a unique identification, moves from one hotel provider to the next, contacts **stationary agents** and collects the **matching hotel data**.

5. The hotel agent finishes its itinerary at the home location of the user.

6. The other mobile agent responsible for the flights that initiated this mobile agent for the hotel data recognizes the hotel agent with the unique identification.

Use Case 3: 'display result'

Actor: -

This use case has a 'uses' relation to use case 1 and use case 2.

Sequence of events

1. The user requested a flight.

2. After the mobile flight agent arrives at the user location again, the matching flight data is displayed in a **table**.

3. The application displays all the necessary information to choose one opportunity.

Alternative Scenario:

1. The user requested a flight and a hotel room.
2. The flight agent and the hotel agent reach the user location after their trip at different times.
3. When the hotel agent arrives, its data is collected in a list.
4. After the flight agent has arrived at the home destination, it looks for its corresponding hotel agent in the list with the unique identification.
5. If no hotel agent is found, it checks all new arriving hotel agents.
6. When the relationship between the two agents can be established, the matching flight and hotel data are displayed in a table.
7. The application displays all the necessary information to choose one opportunity.

Use Case 4: 'Choose one offer'

Actor: user

Sequence of events

1. The application lists all flights and hotel rooms.
2. The user chooses one offer.
3. The **user** enters his or her name and credit card number.
4. A **new mobile agent** moves to the flight and hotel providers of the chosen offer and tries to book the offer. That means that the stationary agents at the provider locations check the available places.
5. In case of a successful order the mobile agent takes a **confirmation** back to the user.

Alternative scenario:

If there aren't any free seats on the requested flight or in the chosen hotel the agent will receive a node about that situation and displays it at the home destination.

The flight will be booked independently of a free hotel room, but a hotel room will only be booked, if the booking of the according flight has been successful.

Use Case 5: 'enter flights'

Actor: flight provider

Sequence of events

1. The flight provider (an airline) enters the flights with a **graphical user interface**. One flight is described with the originating city, the destination city, the weekdays of the flight, the departure time the arrival time, the price and the available seats.

2. The information is stored in a text file.

3. The stationary agent accesses the **text file** to get the latest information.

Use Case 6: 'enter hotel rooms'

Actor: hotel provider

Sequence of events

1. The hotel provider (a hotel chain) enters the locations of the hotels, the total number of hotel rooms, the prices and the contracts between the hotel chain and the airlines with a **graphical user interface**.

2. The information is stored in a **database**.

3. The stationary agent accesses the database to get the latest information.

4.3 Class Modelling

One advantage of the object-oriented paradigm is, that we talk from the beginning of the software lifecycle about these objects, the developer has to implement at the end. The names, structures and relations wouldn't change anymore. Therefore this step is for finding the classes and their attributes with the help of the scenarios. The nouns are extracted from the text of the scenarios, the classes are derived from that extraction and the attributes are determined. The methods of a class are designed in the design phase following after the analysis phase.

4.3.1 Noun Extraction

There are 20 interesting nouns, which are candidates for objects of the application. The object 'table' is part of the main HCI and isn't regarded as a separate object. The objects 'region repository' and 'user destination' are entities offered by the agent system and don't have to be modelled explicitly. 'Flight Provider' and 'Hotel Provider' have just one interesting attribute - the name. That is why they aren't implemented as separate objects.

The following chart shows the remaining objects mapping to a class name. In addition, there are some classes for starting the application and managing control mechanisms.
The objects are collected in three groups. The first group is called the **'entity objects'** and contains data- and store-oriented objects. The **'interface objects'** handle interfaces for the entity objects. The last group is the group of **'control objects'** and comprises objects with a complex processing [Goll97].

Object Description:	**Class name**
Entity Objects	
inquiry data of the user	InquiryFlightData.java
matching flight data	MatchingFlightData.java
collection of outward and return data	FlightOffer.java
matching hotel data	MatchingHotelData.java
user	User.java
flight and hotel data of one request	RequestAnswer.java
Interface Objects	
flight booking system HCI	InquiryInputDialog.java
table for the result, is part of the main HCI	
flight provider HCI	FlightInputDialog.java
hotel provider HCI	HotelInputDialog.java

Contol Objects	
stationary agent of the user	InquiryStationaryAgent.java
mobile agent collecting flight data	FlightInquiryAgent.java
stationary flight agent	StationaryFlightAgent.java
mobile agent collecting hotel data	HotelInquiryAgent.java
stationary hotel agent	StationaryHotelAgent.java
mobile booking agent	BookingAgent.java
entered date handling	OwnDate.java
file access, database access	part of the stationary agents
Entities given by the Grasshopper agent system:	
region repository	
user destination	
Entities with too few attributes:	
flight provider	
hotel provider	

4.3.2 Attributes of the Classes

Relations between objects can be shown in attributes. This section describes all classes with their attributes. The name of an attribute is the name that is used in the implementation phase [GoRa97].

The first group are entity classes, characterized with an emphasis on data. That means that they have more attributes compared to control classes. Interface classes have a graphical user interface.

Class: InquiryFlightData

Name	**Description**	**Type**
outwardDate, returnDate	date of the outward and return flight	OwnDate
outwardWeekday, returnWeekday	calculated week day of the outward and return date, used to search a weekly flight offer	integer
startAirport, targetAirport	city where the flight begins and ends	String
returnFlightRequired	flag, if the return flight is requested	boolean
flightQuantity	the amount of requested seats	integer
hotelRequested	search flag for a hotel room	boolean

This class is the main data object. It holds the request data for the flight and the hotel agent. An object of this class is filled with the entered values of a user.

Class: MatchingFlightData

Name	Description	Type
freeSeats	number of available seats of that flight	integer
flightPrice	price of the flight in Euro	float
departureTime	time when the flight begins	String
arrivalTime	time when the flight ends	String

This class wraps the attributes of a flight offer.

Class: FlightOffer

Name	Description	Type
outwardWay	collection of all flight offers for the outward flight from the start to the destination city	Vector
returnWay	collection of all flight offers for the return flight from the destination to the start city	Vector
provider	name of the airline offering the flights	String

A `FlightOffer` object is used to hold the outward and the return offers. It has two vectors with `MatchingFlightOffer` objects.

Class: MatchingHotelData

Name	Description	Type
flightProvider	name of the flight provider which generated the hotel inquiry agent	String
freeRooms	number of available rooms of the hotel	integer
hotelOfferPrice	price of the room in Euro	float
provider	name of the hotel chain offering the room	String

The flight provider is part of the matching hotel data, because it is used to find the relation between the flight agent and the hotel result at the user's home.

Class: RequestAnswer

Name	Description	Type
flightAllotment	flight offers of one airline	FlightOffer
hotelAllotment	hotel offers of the hotel agent with was initialted of the flight agent at the flight provider location	Vector
offerNumber	street where the user lives, to send the ticket to	integer

`RequestAnswer` is used at the user's home to combine the collected flight offers and the collected hotel offers belonging to the same request. The hotel vector consists of `MatchingHotelData` objects.

Class: User

Name	Description	Type
firstname	Christian name of the user	String
surname	family name of the user	String
cardNo	credit card number of the user to pay the offer	String

An object of this class is generated when the user books an offer.

Class: InquiryInputDialog

In order to get the analysis of this HCI, consisting of the main input dialog and the presentation of the query's result the following drafts were designed. The integrated development environment 'Forte for Java' was used to draw the drafts. These two masks will be put in two tab panels.

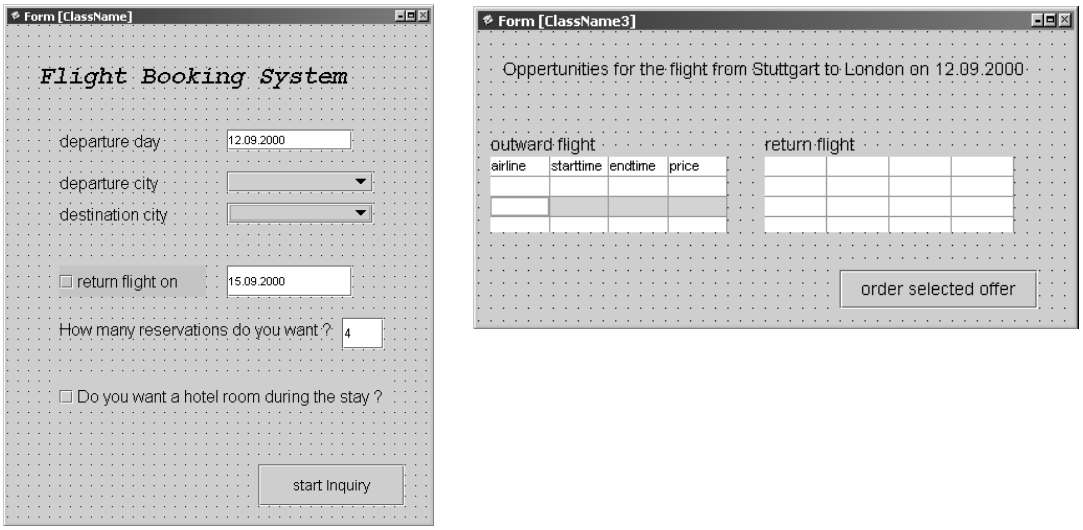


figure 12 dialog draft for the inquiry input dialog

Name	Description	Type
bookingPanel	tab panel with the booking information	BookingPanel
optionPanel	tab panel with the query result	OptionPanel
quiryPanel	tab panel with the input values	QuiryPanel
userPanel	tab panel for the user information	UserPanel
tabbedPane	object that manages all other tab panels	JTabbedPane
actionListener	object that handles all mouse and key actions	ActionListener

The inquiry input dialog is realized with a tab panel consisting of four tabs for the input values, the return result, the user information and the booking panel. This is implemented with four inner classes of the `InquiryInputDialog` class and one inner class for the action control.

Class: FlightInputDialog, HotelInputDialog

These two classes have a very similar functionality. Both HCIs offer the possibility to enter data records for the corresponding information. The following draft shows the graphical user interface for the flight and the hotel providers.

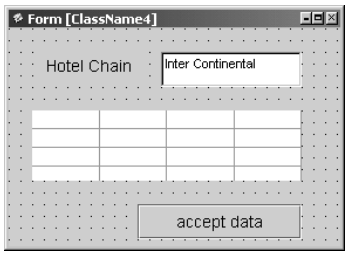


figure 13 dialog draft for the provider input dialogs

Name	Description	Type
providerNameField	name of the company that offers a service	JTextField
providerNameLabel	label for the text field above	JLabel
dataRecordTable	table for the contents of a repository (file for flights, database for hotel rooms)	JTable
acceptButton	button to make the entered Information persistence	JButton

The following Control classes have more functional methods than data classes. These are developed in the design phase.

Class: FlightInquiryAgent

Name	Description	Type
inquiryFlightData	the user's information	InquiryFlightData
producerInfo	agent info of the stationary agent which created the mobile flight agent	AgentInfo
hotelInquiryAgents	if the user requests for a hotel room, this object holds all generated hotel agents	HashSet
flightOffers	all matching data objects of the provider	Vector
agentState	used in the `live()` method to define which code fragment to be executed	integer
agentInfoFlightProvider	array of all providers offering a flight service	AgentInfo[ ]

agentInfoHotelProvider	array of all providers offering a hotel service	AgentInfo[ ]
flightProxy	stationary agent to contact	IStationaryFlightAgent

The FlightInquiryAgent moves from flight provider to flight provider and generates a mobile hotel agent if necessary.

Class: HotelInquiryAgent

Name	Description	Type
inquiryFlightData	the user's information	InquiryFlightData
agentInfoHotelProvider	collection of all hotel services to be visited	AgentInfo[ ]
hotelVector	all matching data objects of the provider	Vector
userHomeIdentifier	identifier to find the way to the user's home	Identifier
hotelProxy	stationary agent to contact	IStationaryHotelAgent
flightAgentIdentifier	identifier of the flight agent to hold a relation	Identifier

A `HotelInquiryAgent` is generated at a flight provider's location and takes the collection of all hotel services to be visited, the `InquiryFlightData` object and the identifier of the flight agent, which initiated it.

Class: StationaryFlightAgent

Name	Description	Type
data	one record that conforms to the user data	MatchingFlightData
flightData	request data of the user	InquiryFlightData
dataList	all flight data objects to compare with the user data	Vector
fileIn	data file name with the flight information	String
requestAnswer	object with the offer to book	RequestAnswer
user	user who wants to book a offer	User

This agent has a file and the booking information as its attributes. The functionality is checking the flight availability and booking a flight.

Class: StationaryHotelAgent

Name	Description	Type
connection	connection to a database	Connection
stmt	statement, which is sent to the database	Statement
result	result set of the statement	ResultSet
requestAnswer	object with the offer to book	RequestAnswer
user	user who wants to book a offer	User

flightData	request data of the user	InquiryFlightData

This agent has a connection to a database and the booking information as its attributes. The functionality is checking the hotel availability and booking a hotel room.

Class: StationaryInquiryAgent

Name	Description	Type
queryDialog	graphical user interface to enter the values	InquiryInputDialog
inquiryAgentProxy	proxy object of the generated flight agent	IFlightInquiryAgent
flightVector	all matching flight offers	Vector
hotelMap	map containing the flight providers as the key and the vector of matching hotels as the value	Map
bookingAgentProxy	proxy object of the booking agent	IBookingAgent
answerMap	map of RequestAnswer objects that contain the flight and hotel return information	HashMap

This is the most important class of the application. It starts the inquiry input dialog and handles all arriving agents at the user's home. It has an `answerMap` with all answers of a request, a `hotelMap` with hotel offers and the `flightVector` with flight offers.

Class: BookingAgent

Name	Description	Type
requestAnswer	object with the flight and the hotel to book	RequestAnswer
successfulFlightBook	result of the flight booking	boolean
successfulHotelBook	result of the hotel booking	boolean
hotelProxy	stationary agent to contact	IStationaryHotelAgent
flightProxy	stationary agent to contact	IStationaryFlightAgent

This agent moves to one flight provider and one hotel provider to book the offer. The two boolean values indicate if the booking request was successful or not.

Class: OwnDate

Name	Description	Type
date	object in the Java date format	GregorianCalendar

This class takes a string and converts it to a Java date format.

4.4 Discussion

It is important to offer the user actually generated information about the user's request. To provide the latest data records is one of the features of this application, because it always has direct access to the data repository. There are two possibilities to handle a situation, where the user wants to request different offers and decides which one to book a certain time later.

The problem is, that an undetermined number of users can access the data of a provider. If someone requests the available seats of a flight, this person will not decide to book the flight at the same moment. First all offers are collected and some minutes later the user initiates the action to book one. That means that the situation of the available seats can be very different to that when the request was made some minutes ago. The user cannot trust the information regarding the number of free seats, because the number can be obsolete. How can that situation be resolved?

The two solution possibilities are the pessimistic and optimistic data handling. The following description is mentioned in the requirement list. If the stationary agent of a provider finds matching data, it reduces the requested flights from the available, because the user could choose this offer after he or she has compared it with the other possibilities. That means a reduction takes place independently of whether the offer is taken or not. The advantage of that mechanism is, at the time the user compares all the offers, he can rely on the system, that the offer he will take is pre-booked for him or her and he or she gets it in any case. The big disadvantage is, that all providers reduce their available places, although the user just takes one offer. There is a distortion of the real booked flights and those just requested. After the user has chosen one offer the provider will get a confirmation, but the other providers who reduced their number of free seats won't get the information that the user won't take their offer. Another disadvantage is that a countdown is needed to increase the available flights with the requested number of seats after a special time with the assumption that the user took another offer. That is needed to release the pre-booked seats.

The second technique to handle the request for a number of free seats is the optimistic way. This one is easier because the stationary agents of the providers tell the mobile agents the available flights at the time of the request. There is no reduction or pre-booking and the system has always the real number of free seats. When the data is displayed to the user, he cannot rely exactly on the data, because another customer could have changed it at the provider. At the time the user books one offer, the specific provider gets a booking request. The current

available seats are checked again and the process is executed. If the requested number of seats is not free anymore, the agent takes back a note to the user.

The advantage of the second implementation is that there won't be a decrease and increase of the number of available places. The administration with the timeout and the pre-booking is too complex for the single advantage of the reliable booking possibility. The second technique is more state-of-the-art and it is used in this project.

4.5 The Object Diagram

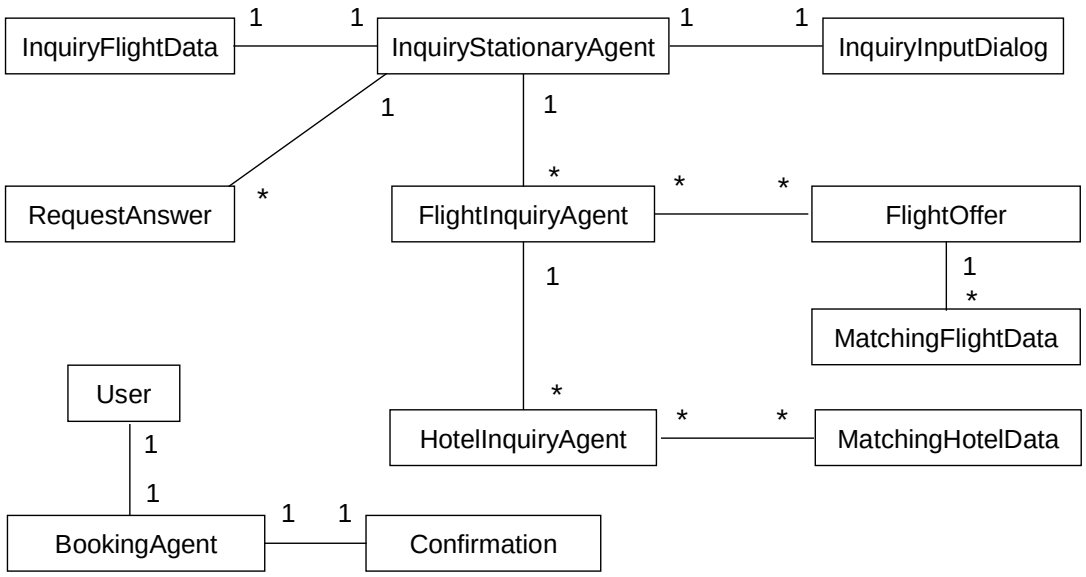


figure 14 object model of the Flight Booking System

The object diagram shows the relations between the objects. There are three kinds of relationships. The one-to-one-connection describes that one object has one instance of the other object. With the one-to-many-connection, one object can have multiple objects of the other class. The third version is the many-to-many-connection to show that both objects can possess more than one instance of the other class. A star on a line indicates that one object can have many objects of the other one.

An `InquiryInputDialog` has a one-to-one-connection to the `Inquiry-StationaryAgent` that can have one or more `FlightInquiryAgent`. Every new request of the user generates a new `FlightInquiryAgent` but one `FlightInquiryAgent` always has just one `InquiryStationaryAgent`.

One `InquiryStationaryAgent` has one `InquiryFlightData`, consisting of the information that the user entered. This data object is passed to every mobile agent. Starting a new request means instantiating a new `InquiryFlightData`. The `FlightInquiryAgent` has several `MatchingFlightData`, because many flights can be conform to the `InquiryFlightData`. The `MatchingFlightData` objects are wrapped in a `FlightOffer` object. There are two vectors for the outward and the return flight. The `HotelInquiryAgent` is in the same situation. One `FlightInquiryAgent` can have more than one `HotelInquiryAgent`; every new flight offer of an airline generates a new hotel agent. The `BookingAgent` comprises a user and a confirmation.

4.6 The Class Diagram

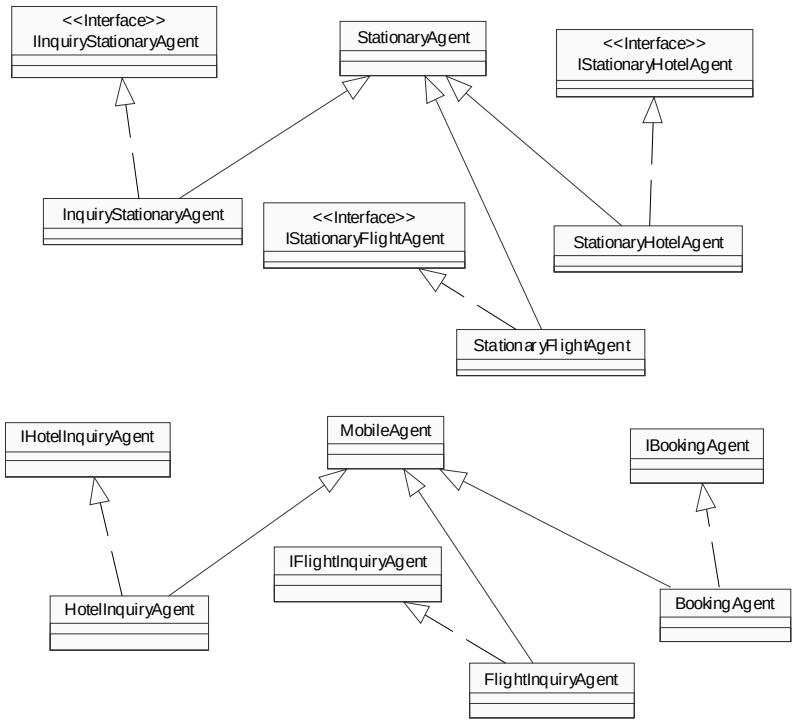


figure 15 class diagram of mobile and stationary agents

The class diagram in figure 15 shows three stationary and three mobile agents. Every kind inherits from the corresponding core class of the Grasshopper agent system. The class `StationaryAgent` is the base class of all agents, which are not allowed to move and the class `MobileAgent` is deployed, if agents want to move on their own purpose. The interface of every agent class is used for the proxy generation. The method `newInstance()` of the class `ProxyGenerator` generates an instance with the interface class. Only the methods of the interface can be invoked.

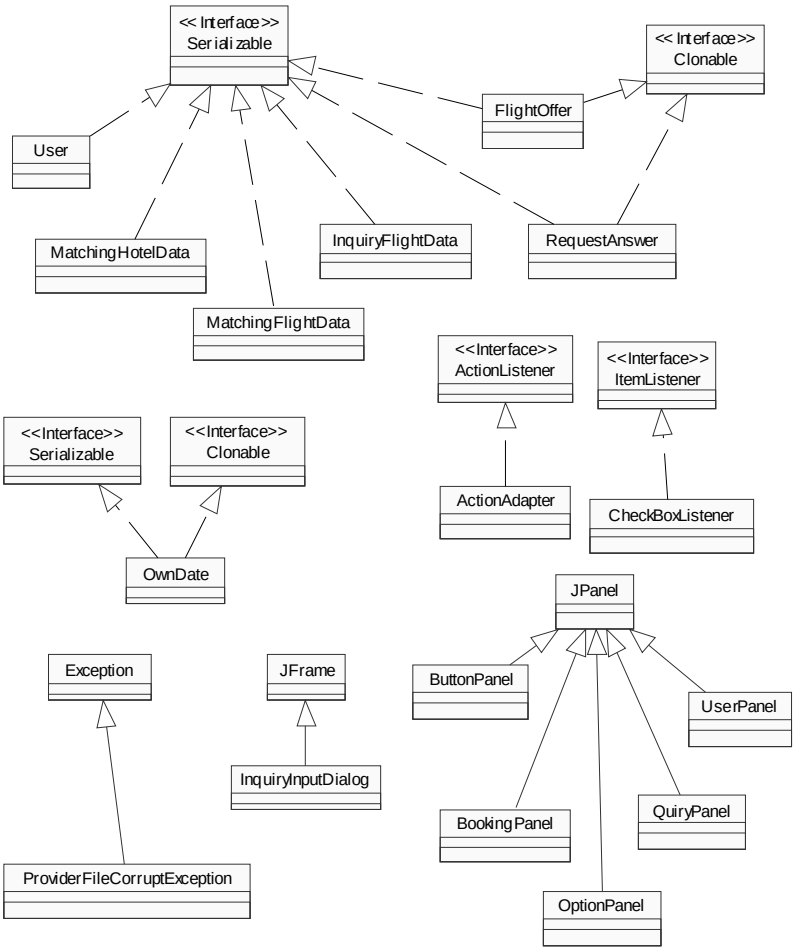


figure 16 class diagram of data and GUI and utility classes

The class diagram above shows the classes of the graphical user interface and the data classes. All data classes implement the interface `Serializable`. The reason is, that all data objects are serialized when they move with the mobile agent. `FlightOffer` and `RequestAnswer` are `Clonable`, because objects of these classes are changed and a clone object has to be used to keep the values of the origin object. If the object isn't cloned, all references of the object are changed. The different Panels, derived from `JPanel`, constitute the tabs of the inquiry input dialog. The `OwnDate` class is introduced to parse the entered values of a date and to calculate the weekday. A `ProviderFileCorruptException` is thrown if the access of a file at the flight provider failed.

4.7 The Data Model

The noun extraction from the scenarios was useful to find the objects that have to be modelled. The objects have been divided in three groups. The entity classes are data oriented. These classes are part of the data model. This model shows the relation and the connection of the data classes.

Inquiry Flight Data
- outward date
- return date
- start city
- destination city
- return flight required
- flight quantity

User
- card no.
- first name
- surname

figure 17 data model - outward data

These two entities are taken from the mobile agents to execute the search or the booking. The inquiry flight data entity and the user entity are filled with the entered data of the dialog. The inquiry flight data is used for the request at the flight and at the hotel provider.

The stationary agents of the providers generate the matching data with the inquiry flight data. These matching data is put together to a request answer entity at the stationary inquiry agent at the user's home, because the mobile agents arrive at different times at the destination host.

A request answer entity is shown in figure 18. It consists of three entities; one for the flight part, another for the hotel part and the third is the offer number. The flight part has two vectors of matching flight data, one for the outward and the other for the return flight. The matching hotel data entity has the flight provider as an attribute, because it is used to calculate the price of a room and to combine the hotel data to the corresponding flight data.

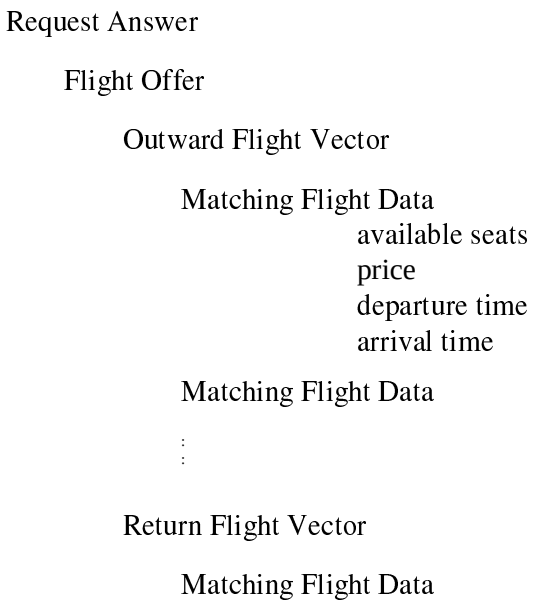


figure 18 data model - return data

The request answer entity of figure 18 is a nested data object. Other data objects that are independent are part of it (Flight Offer, Matching Flight Data). The situation that one entity is part of another is typical for a data model. These data can be described with the extensible markup language XML. It is a document type definition to standardize own data. It is a suitable way to model data and their relations. A XML document consists elements built of attributes or other elements. Every entity can be transmitted and exchanged because of the common structure. A style sheet document forms the visualization of a data object. The separation of content and style is an important topic in XML.

5 How to manage the Design Phase?

To design architecture means to put the deployed classes of the analysis phase in a suitable structure to reach the constraints of the project. After the structure is found, the communication will be designed, including the channels between computers and the messages between entities [Hru99]. A top-down design begins with the architecture of a system and ends with the methods of the classes. The software design phase consists of three activities: architectural design, detailed design, and design testing. From the viewpoint of abstraction, during the architectural design the existence of certain modules is assumed. Then the design is developed in terms of those modules.

The analysis phase is completed, if every possible scenario can be executed with an interaction of the involved objects. All requirements have to be covered by a use case and all classes are found with the help of the scenario descriptions. The input of the design process is the specification document, a description of *what* the product is to do. The output of the design process is the design document, a description of *how* the product achieves the specification.

5.1 System Design

The first step in the design phase is the development of the system architecture. A general and strategic view on the solution is manifested and documented according to the overall structure and the character of the system.

The objective of the design is the system architecture, the building of subsystems and the context and realisation of solutions.

5.1.1 Overall Structure

The Flight Booking System is a distributed system. It is linked with a local area network or over the Internet. A central machine administers the services of every included computer. The whole system is divided into subsystems. One is the system for the user, another one is the part for the two kinds of service providers and the last section is the region registry location machine. (see appendix)

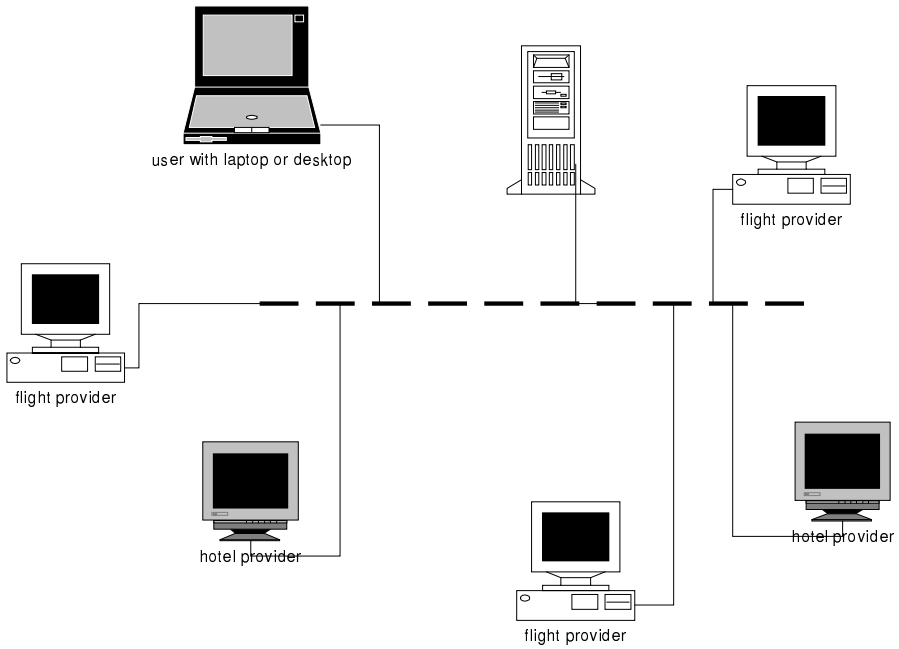


figure 19 overall design of the Flight Booking System

Figure 19 shows a typical architecture of the Flight Booking System. The dashed line in the middle can be any network. That means a connection between two computers, a local area network or a wide area network including the Internet or an intranet of a company. The region registry is resident in one computer of the network. It has a list with all the services offered by the system. The different providers are linked to the same network and register their services at the registry. The user of the application just connects to the network and executes his or her request. Therefore any kind of computer can be used – a desktop, a laptop or another mobile device.

5.1.2 Subsystems

A subsystem is a mandatory sub component of the total system. All subsystems build a closed task or a problem domain. A subsystem has a responsibility that is described by **functionality**, a **place** or location and the **relations** to other subsystems. It consists of classes, interfaces, associations, operations and results [Hau98]. The separation of the system is organised by horizontal layers and vertical segments with the consideration of the constraints of platforms, operating systems and middleware software. Figure 20 on the next page shows another view on the system. The agent system is the middleware and runs on every machine that is involved in the system. The ‘Grasshopper’ agent system is available for different operating systems. The agent system hides the operating system and it is not important which one is used.

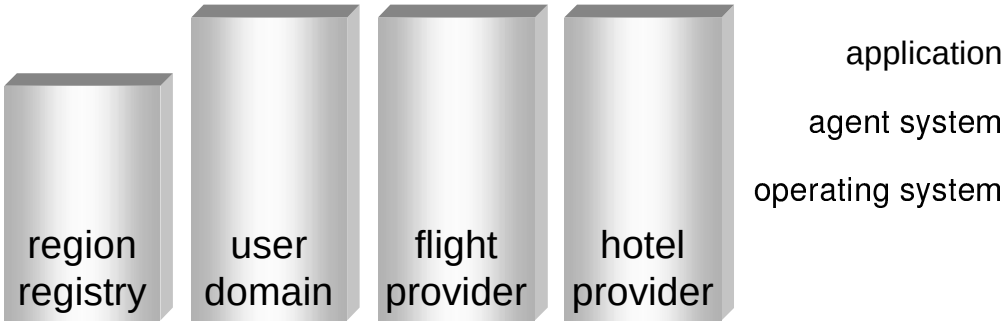


figure 20 design of the subsystems

The Flight Booking System project has four subsystems:

- One subsystem for the user desktop to enter the request

The *functionality* of this subsystem is the possibility to enter the inquiry data about the flight and the hotel booking. It also includes the start of the request, the visualization of the result and the booking of the chosen offer. It is *located* on every computer of a user. It just needs the agent system and the application part for the user input. This subsystem has *relations* to all other parts. It has a relation to the region registry subsystem to get information about the services and a relation to the subsystems of the providers when an agent requests for available flights or hotel rooms.

- One subsystem for the flight provider

The *functionality* of this subsystem is to answer a request of a mobile agent. It accesses a data file of the flight data and responds with the matching flight data. The *location* is the computer of an airline. An agent system, the data information, the stationary agent of the application and the user interface to enter flight data has to be installed on this computer. This subsystem has a *relation* to the region registry and a loose connection to the user interface subsystem when it serves a request.

- One subsystem for the hotel provider

The *functionality* of this subsystem is also to answer a request of a mobile agent. It accesses a database of the hotel data and responds with the matching hotel data. The *location* is the computer of a hotel chain. An agent system, the database, the stationary agent of the application and the user interface to enter hotel data has to be installed on this computer. This subsystem has a *relation* to the region registry and a loose connection to the flight provider subsystem when it serves a mobile agent, which is initiated at a flight provider if matching flights are found.

- One subsystem for the region registry

The *functionality* of this subsystem is to administer the services and to offer a list to all providers. The *location* is a computer with the agent system and doesn't need a special installation. It has *relations* to all other parts.

The interfaces have to be small. The interfaces of this project are only the agents moving from one subsystem to the next.

A component represents a part of the implementation of the system. Components can be source or binary code, documentation files, executables and so on. A component diagram shows the structure of the source code.

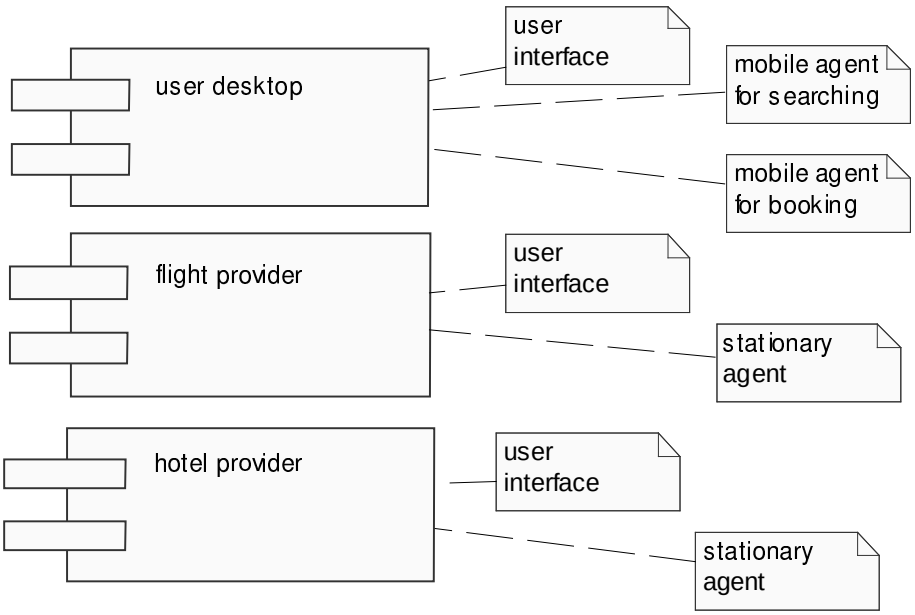


figure 21 design of the components

The component diagram of figure 21 shows the three components and the corresponding elements containing the graphical user interface and the different agents.

5.1.3 Constraints of the Overall System

Another part of the system design is the decision about class libraries and tools. This section summarizes the definitions and results of the requirements, the use cases and the scenarios. The topics of that step are constraints and influence the design phase.

5.1.3.1 User Interface

The main user interface is the one for entering the inquiry data. The other two are used to provide the available flight and hotel information. All are developed in Java. The Swing standard class library is used to derive the interface classes of the Flight Booking System. Reusability is an important feature. The interactions of the user are clear because of an intuitive handling.

5.1.3.2 Persistence

Persistent objects live longer than the actual application runs. That can be achieved by storing the data in the file system or in a database. Normally the lifetime of an object is volatile, it only exists during the execution of the application. To make an object permanent, special actions have to be initiated.

The two design possibilities are used in this project. The flight data is stored in a file and the hotel data is stored in a relational database management system. The GUI of the flight provider generates an ASCII file, which can also be edited with a text-processing tool. The database is an Oracle Database Version 8i running on every hotel provider host. The Java Database Connectivity JDBC API is used to access the database with the structured query language SQL. The object design explains the structure of the file and the relations of the database.

JDBC establishes the connection to the database, sends the SQL statement and handles the result set. A driver manager takes the transparent access to the ODBC Bridge of the operating system and is responsible for the network driver. JDBC commits or rolls back the transaction.

Streams are the only way to store objects file-oriented. The class library offers a File class to serialize the information.

5.1.3.3 Operating System

Java is a platform independent language. It can be used on every machine for which a virtual machine exists. The ‘Grasshopper’ system is released for Microsoft Windows and for Unix. The test environment for this project consists of Intel machines with Windows NT 4.0.

5.1.3.4 Hardware

The chosen hardware is always in conjunction with the operating system. Because of the Windows platform an Intel computer or a similar i386 is used. Of course it is also possible to use an IBM, Sun or HP workstation.

5.1.3.5 Middleware

The 'Grasshopper' agent system is used as the middleware[3]. The middleware handles the registration, localization, communication, migration, heterogeneity, scalability and the complexity of the network for the distributed objects. A central server organizes all the traffic. Other middleware standards are CORBA the Common Object Request Broker Architecture, the distributed component object model D/COM, Java Beans, remote method invocation (RMI) or the hypertext transfer protocol HTTP used between browsers and web servers.

5.1.3.6 Class Libraries

Because of the deployment of the programming language Java, the class library question is answered. The Java Development Kit Version 1.3 and the application programming interface of the 'Grasshopper' agent system offer all relevant classes. The object oriented concepts like inheritance and polymorphism of the language will be used.

5.1.4 Distributed Systems, Communication

The main effort of distributing a system is to separate the application intelligence onto different systems. The client/server paradigm is one possibility. A comparison with the agent technique has shown that just one kind of programming architecture[4] (client/server or mobile agents) could be used.

A distributed system also means the use of threads. Some processes can be executed in parallel. If a matching flight data is found the flight agent initiates a mobile hotel agent and both work in parallel. These are active objects because they throw events, initiate actions, control threads and produce data. A result of this is synchronization at the home location of the user. Both agents have to find each other again.

Communication Concepts

In the context of 'Grasshopper', inter-agent communication may be performed in several modes. Agents sometimes want to transfer information to another agent via a proxy. Communication includes a server offering a service and a client requesting for something. 'Grasshopper' supports the following communication modes:

[3] section 2.2 Grasshopper
[4] Interim Report section 2.2.3 Client/Server versus Mobile Agents

5.1.4.1 Synchronous Communication

When a mobile agent invokes a method on a server, or a method of another agent, the server executes the called method and returns the result to the client that continues its work then. This style is called synchronous because there is no parallel work and the client is blocked until the result of the method is sent back.

5.1.4.2 Asynchronous Communication

The disadvantage of synchronous communication is, that the client, and the respective starting agent are blocked and it has to wait for the answer. With the asynchronous version the client continues performing its own task. There are several techniques for the client to get the result of the invoked method. It can periodically ask the server whether the method execution has been finished, or subscribe to be notified when the result is available.

The first version is called ‘polling’. After an application has started a mobile agent, it looks periodically to see if the agent is back at the home destination. If the agent is back, the application will find the agent and it can be used for further execution. The second version is called ‘call back’. The sender application subscribes at a result storage place that it is interested in the result of a receiver. If the mobile agent comes back to the home destination, it will be stored at the storage place. The place notifies all entities that subscribed to get information about a new agent and the senders can request for the new agent.

5.1.4.3 Multicast Communication

Multicast communication enables clients to use parallelism when interacting with server objects. By using multicast communication, a client is able to invoke the same method on several servers in parallel [ikvPG98].

Group proxies provide a framework for designing and implementing partitioned concurrent activities in Java. A group consists of an arbitrary number of members. Group proxies maintain some kind of collection by enabling objects to join and leave groups dynamically. They also encapsulate the thread-based mechanisms needed to implement Java analogies of execution constructions found in concurrent, parallel languages. Internally, concurrent communication is implemented by the creation of parallel running threads where each of them performs a specific communication task. For this purpose, the group request is split and addressed to the appropriate server in the first phase, the so-called scatter phase. In the case where all the threads perform invocations without return values, only the scatter part firing up the tasks applies. But when actions must be synchronised or results must be collected, a co-termination policy is required.

5.1.5 Error and Exception Handling

This topic is very important because the application has to be fault tolerant. If an exception occurs, a planed and structured system reaction has to be executed. This begins with the input of the user interfaces. All text fields should verify their input and communicate with the user.

Risky areas and situations should be identified. These are the migration of the agents, the access to the data repositories and the movements back to the home destination. Perhaps a recovery mechanism helps to achieve a previous state again.

5.1.6 Start up and Shutdown

Every location that is involved in the application has to start the agent system. The initialisation with the start up includes the starting of the system, establishing the environment, registration at the region registry. This also includes the start of the database. The shutdown or termination ends the appropriate part of the application and the agent system.

5.1.7 Parallel Tasks

The Flight Booking System is a multi-user application with concurrent actions. Simultaneous access is possible. Therefore the emphasis on synchronization and access control is necessary for the file access at the flight provider and the database access at the hotel provider. The agents migrate independently of each other. They execute in parallel and come back to the home destination, where the information will be collected.
Parallel tasks are performed with active objects like threads or processes. These objects possess their own control flow and can initiate activities. Passive objects contain data and cannot trigger an activity. Synchronisation is necessary when data is written by different objects and can be realized with a semaphore or the Java synchronized statement.

5.2 Object Design

In this section the details of the objects will be modelled. The **constraints** of the analysis model and the **decisions** of the system design influence the solution of every class. [Hau98]

The following thoughts take the results of the system design and handle the possible realization needs. The first step describes the design guidelines. These comprise information hiding of internal data, reusability, complexity, interfaces and correctness. The modularity is very important – there should be low coupling with regard to the use of foreign classes and high cohesion between classes of a subsystem - realized with inheritance. The second step is the refinement of the analysis model considering the methods and if necessary adding new attributes. After that stage the chosen class library of the system design has to be checked if all structures are available according to inheritance, association and aggregation. The fourth step is the realization of the persistence constraints. If necessary the last step consults additional classes of the operating system API. Some checks have to prove whether an API helps to solve tasks of the project.

5.2.1 Methods of the Classes

The following classes were found in the analysis phase. The attributes are still defined is the analysis section. The methods are described in this section in conjunction with the sequence diagrams. Sequence diagrams for each use case help to find the methods of a class. These diagrams show the interactions of the objects in the temporal order top down. Every object has a lifeline that is the starting point for the messages sent to other objects. The requirements for the sequence diagrams are scenarios, representing the actions in textual form. The methods listed in this section are the most important ones. The list isn't complete, but it shows the messages needed to perform the requirements. The set and get methods for each data attribute aren't part of this listing.

Use Case 1: Request a flight

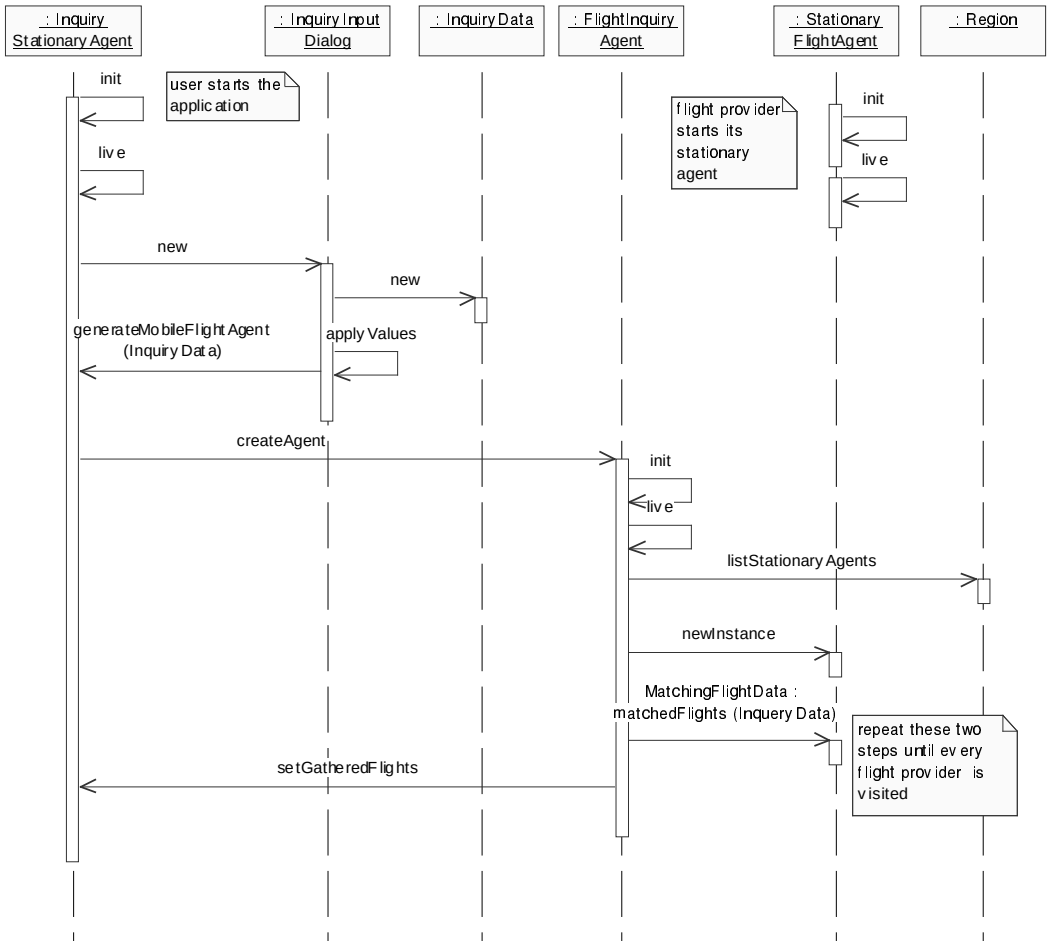


figure 22 sequence diagram for requesting a flight

Every stationary and mobile agent has an `init()` and a `live()` method. The `init()` method is executed once when the object is initiated and the `live()` method is invoked every time the agent migrates. The `InquiryStationary-Agent` generates an `InquiryInputDialog` to enter the `InquiryFlightData`. After the data is verified the `InquiryStationaryAgent` starts a mobile agent to search for the requested information. The `FlightInquiryAgent` takes a list of destination hosts from the region registry and contacts the `StationaryFlight-Agents`. The matching flight data will be returned and taken from the mobile agent. After all hosts of the itinerary are visited, the mobile agent transfers the gathered flights as a `FlightOffer` object to the `InquiryStationaryAgent`. An example of the movement is shown in the appendix.

Class: InquiryStationaryAgent

Name	Description
init	Start-method of the application. Initiates the input dialog.
generateMobileFlightAgent	This method takes the InquiryFlightData and starts the FlightInquiryAgent. It requests the current region registry.
live	Shows the input dialog.
setGatheredFlights	Sets the set of matching flight data.
getCheapestOfferString	Calculates the cheapest offer.
showMergedResult	Generates the result to display it.
action	Merges the flight and the hotel result.
startBooking	Generates a mobile booking agent.

There are methods to generate the mobile agents for the request and the booking. `ShowMergedResult()` constitutes the text of the offers with the flight and the hotel result and displays it. The benefit of the application is to find the cheapest offer. One method calculates the cheapest offer.

Class: InquiryInputDialog

(constructor)	Show main panel with input pane and assign listeners to that class.
applyValues	Verify the entered values and wrap it in an 'InquiryFlightData'-object.
innerclass: ActionListener, actionPerformed	The 'ok' button of the panel has registered an action listener and calls the `actionPerformed()` method, which calls `applyValues()`.
setReturnText	Displays the offers in the result panel.

This class handles all tasks to enter and display data. It verifies the data with the `applyValues()` method and has seven inner classes to display and choose the offers and the request data. The inner classes are the panels of the tab panel. They are shown in the class diagram of figure 16.

Class: InquiryFlightData

Name	Description
equals	Compares two InquiryFlightData objects if they are equal.

This class is data oriented. It is passive and doesn't have any control influencing methods. It is used to carry information and is part of the entity object group. The methods of this class are only set and get methods of the attributes.

Class: FlightInquiryAgent

Name	Description
init	Takes the inquiry data.
live	Takes the hotel providers from the region registry and travels to the providers to contact them.
getAgentInfoFlightProvider	The flight inquiry agent provides the providers with the flight service with the help of the region registry.
getAgentInfoHotelProvider	All providers with the hotel service.
generateReturnFlightData	Takes the entered data and changes the departure and destination cities to request the stationary flight agent again.
getAgentState	The agent state defines which section of the `live()` method to execute.

Class: StationaryFlightAgent

Name	Description
init	Initialises the flight data.
live	Assigns the delivered flight data to the local object.
findFlights	Calls `evaluateLine()` and returns the MatchingFlightData.
evaluateLine	Compares the possible and the requested data.
evaluateWeekdayPattern	Maps the entered day to the corresponding weekday.
bookOffer	Stores the user's data in the bookings.dat file.
checkAvailability	Checks if the number of flights are available.

Class: MatchingFlightData

The methods of this class are only set and get methods of the attributes.

Use Case 2: request a hotel

The user can only request a hotel, if a flight exists. After matching flight data is found, the `FlightInquiryAgent` generates a `HotelInquiryAgent` and passes the list of hotel providers to it. When the mobile agent arrives at a hotel provider it contacts the stationary agent and sends a message to it. The `StationaryHotelAgent` takes the `InquiryFlightData` and returns a `MatchingHotelData` object. After the mobile agent has visited the last hotel provider, it returns to the user's home location and adds the gathered hotel information to the stationary agent of the user.

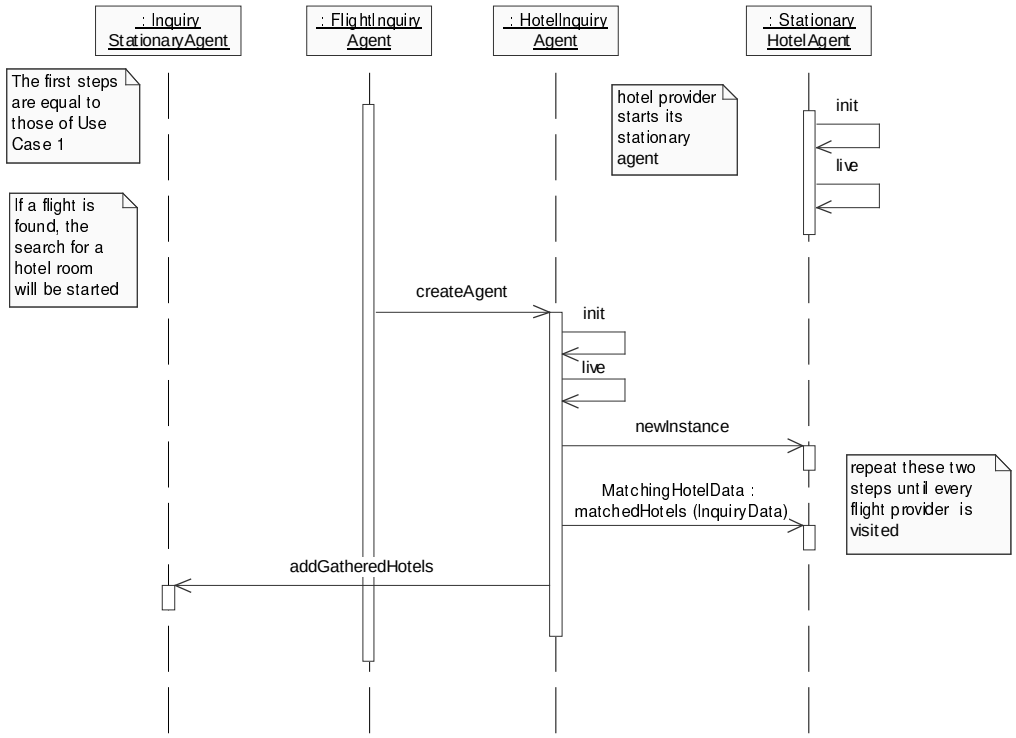


figure 23 sequence diagram for requesting a hotel

Class: HotelInquiryAgent

Name	Description
init	Makes standard settings for the agent.
live	Moves through the network and calls the `matchedHotel()` method.
getHotelVector	Returns the results of the free hotel rooms.
getFlightProvider	The flight provider is used to look if there is a reduction because of a contact between the hotel chain and the airline.

Class: StationaryHotelAgent

Name	Description
matchedHotel	Compares the requested data with the available information and executes the SQL database statements.
init	Generates an object of HotelAgentData and a database connection.
live	Initialises the data.
bookOffer	Checks the availability and stores the user's data in the database.
assignHotelProvider	Read the hotel provider name and the data for the database connection out of a properties file.

Class: MatchingHotelData

Name	Description
equals	Compares matching hotel data objects.
getHotelPrice	Returns the price of a hotel room.
setHotelPrice	Sets the price of a hotel room.

This class also includes the set and get methods of all other attributes.

Use Case 3: display result

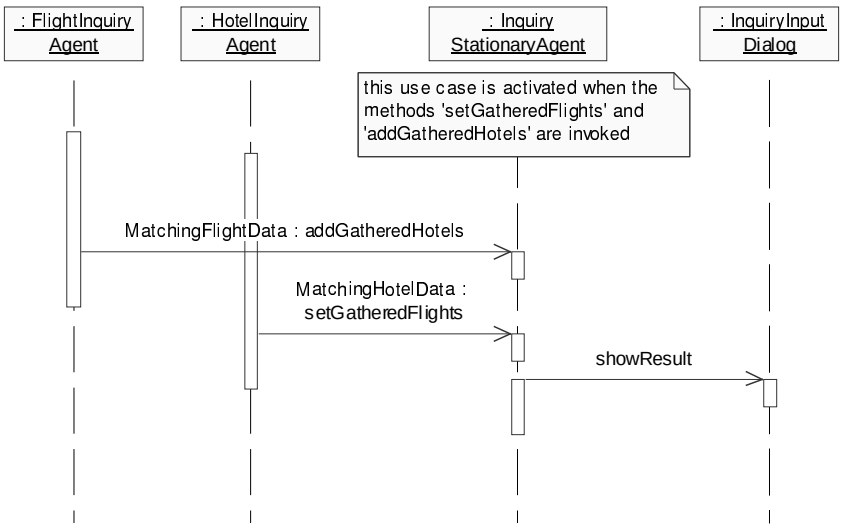


figure 24 sequence diagram for displaying the result

The `InquiryStationaryAgent` collects the incoming agents and puts the related flight and hotel agents together. After that it starts an input dialog to show the flight and hotel data with all the attributes.

Use Case 4: choose one offer

The user selects one offer of the input dialog and presses the button to book the offer. The application calculates the cheapest offer and the user can book it with a single click. A booking agent will be generated, which takes the individual information of a user and migrates to the provider. The stationary agent checks if the offer is still available and books it. If all possible places are sold, the confirmation receives a note about that. (see appendix)

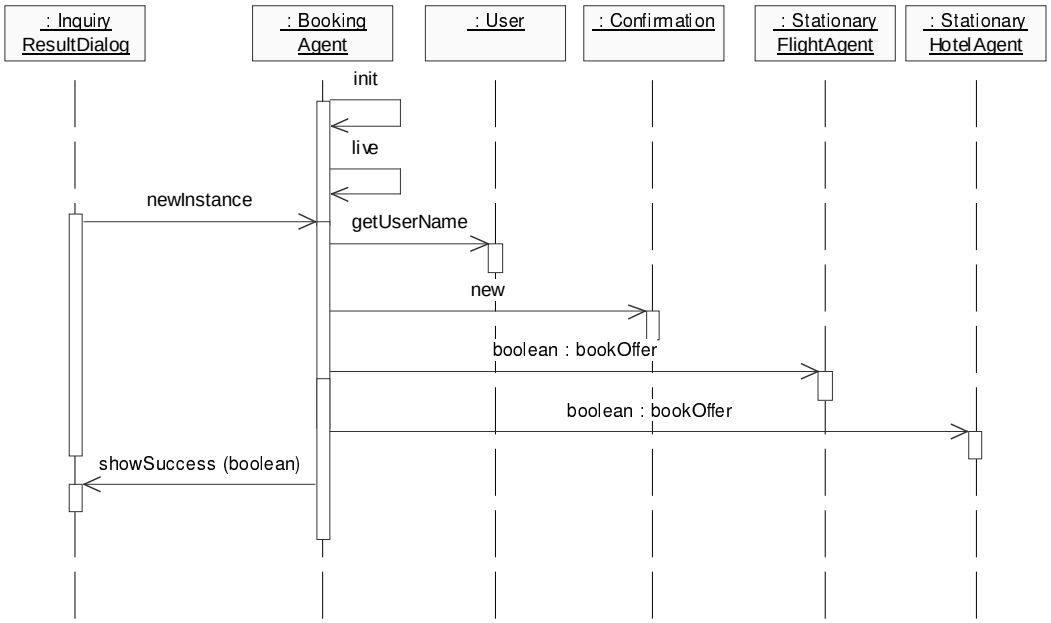


figure 25 sequence diagram for choosing one offer

All stationary agents have a `bookOffer()` method to provide the booking service. It returns a boolean value which indicates the success or the failure of the booking task. The `live()` method of the `BookingAgent` takes this return value and initiates a dialog, which tells the user the contents of the confirmation.

Class: User

Name	Description
getFirstName	Returns the name of the user.
getCardNumber	Returns the credit card number to book the offer.

Class: BookingAgent

Name	Description
init	Sets up the agent.
live	The agent moves to the providers of the selected offer and calls the `bookOffer()` method to contact the stationary agents.
setRequestAnswer	The `RequestAnswer` object contains the data of the chosen offer.
homeMove	Moves the agent to the user's home.

Use Case 5: enter flights, Use Case 6: enter hotel rooms

These two use cases are only actions to fill the data sources with information. They only consist of a user interface and a storage method. The sequence diagram is very small.

5.2.2 Moving Data

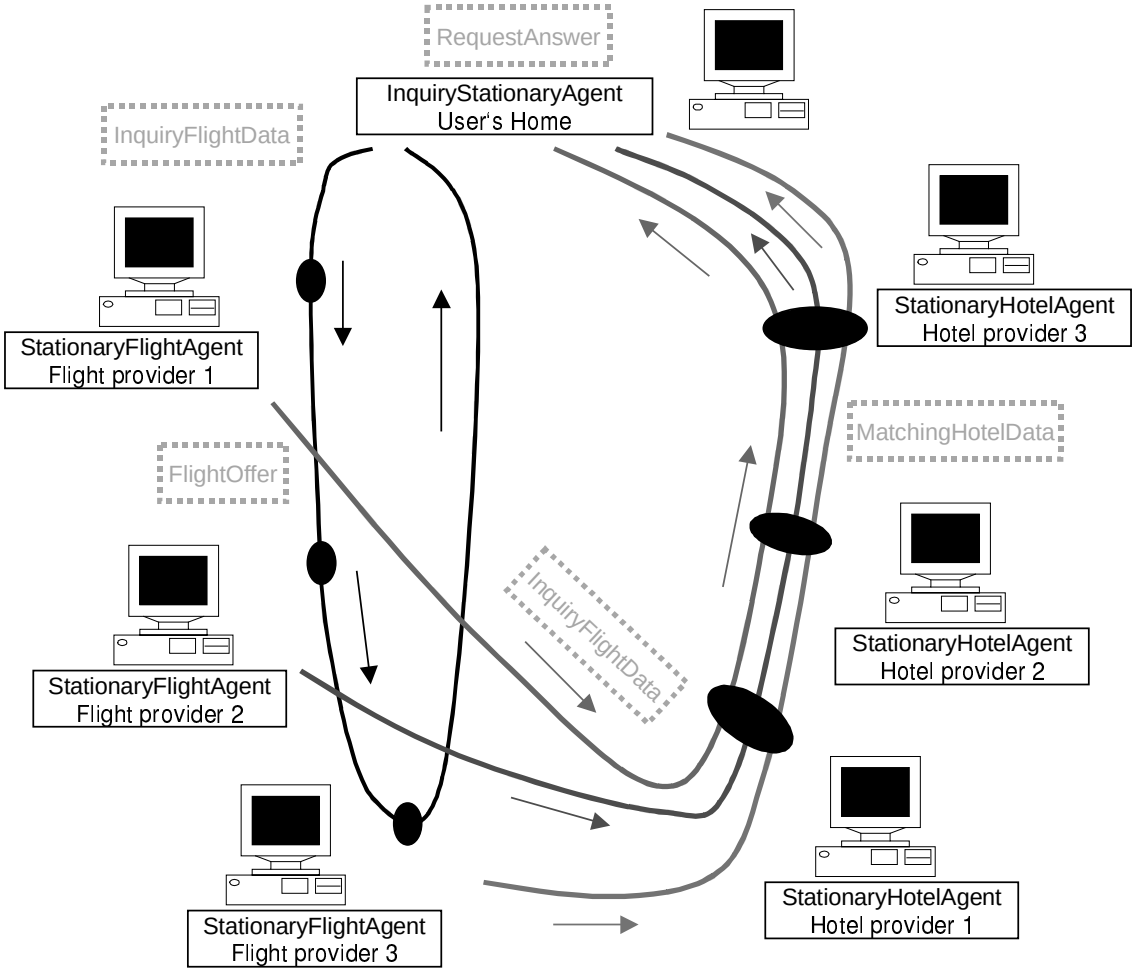


figure 26 movements of objects and data

The figure above shows a black arrow for the `FlightInquiryAgent`, which holds the `InquiryFlightData`. There is one flight inquiry agent per request execution. The black line leads to all flight providers and back to the user. If the search for a flight was successful and the user requested a hotel room, a hotel agent will be generated. The blue, red, and green lines symbolize that. All hotel agents move to the hotel providers and check the price of a room with the information of the flight provider. A `FlightOffer` object is filled with the `MatchingFlightData` for the outward and return way. The hotel agent collects

all `MatchingHotelData` in a vector. At the user's home the results of the flight and the hotel agents will be merged and wrapped in a `RequestAnswer` object.

5.3 Persistence Realization

There are two persistence possibilities to store data. Both are realized in this project. The flight data is stored in a file. One file has the timetable of an airline, one holds the available seats of a flight and another includes the booking user information. The hotel information is stored in a database.

5.3.1 File Structure

Every flight provider generates with the user interface a file with the name `flightdata.dat` and stores all offered flights in it. The flights take place weekly. One data record of a flight consists of the start location, the destination location, the weekday, the departure time, the arrival time, the total seats of the flight, and the standard price of one flight.

There is a special code for the weekday. This pattern can consist of the number of the weekday or 'X' or 'Xe'. If the weekday pattern is 'X', the flight will be served daily. If the symbol is a 'Xe' followed by a number, the flight will be served every day except of the indicated weekday. 'Xe5' means every day except of Friday. If the weekday pattern is a number, the flight takes place on those days. The first day of the week is Monday.

```
# ---------- file  flightdata.dat ----------
Stuttgart#Berlin#1#6:00#7:30#170#190.00
Stuttgart#Berlin#Xe67#7:15#8:45#183#180.00
Stuttgart#Berlin#X#17:00#18:40#188#175.00
Stuttgart#Berlin#37#21:30#23:00#160#200.00
Berlin#Stuttgart#1#7:00#8:25#210#205.00
Berlin#Stuttgart#X#8:15#9:30#175#185.00
Berlin#Stuttgart#Xe24#18:15#19:45#185#195.00
Berlin#Frankfurt#2345#7:40#8:35#200#115.00
# ----------------------------------------
```

figure 27 all flights of one flight provider in a timetable structure

If somebody has booked a flight, the available places will be reduced. The number of seats in the `flightdata.dat` file is the total number of places in the airplane. The `availability.dat` file includes the current available seats of a special flight on a special day. The booking agent manipulates the availability file.

```
# ---------- file  availability.dat --------
Frankfurt#Berlin#1.1.2001#9:50#3
Stuttgart#Berlin#1.1.2001#6:00#20
Stuttgart#Berlin#1.1.2001#17:00#2
Berlin#Stuttgart#4.1.2001#8:15#7
Paris#Stuttgart#2.2.1999#9:00#9
# ----------------------------------------
```

figure 28 every single flight with the available seats

The following file is the most important file for a flight provider because it includes the users who ordered a flight and the invoices are generated with that information. The providers can retrieve from that information who ordered a flight on which day and how much everyone has to pay.

```
# ---------- file  bookings.dat ------------
James  Taylor  4123412341  Stuttgart  Berlin  1.1.2001  7:15  4.1.2001  8:15  350.0
Mike  Hall  2321234534  Stuttgart  Berlin  1.1.2001  7:15  4.1.2001  8:15  350.0
Joe  Peterson  5532786410  Berlin  Frankfurt  20.2.2001  7:40  115.0
Linda  Rush  9879921570  London  Stuttgart  2.2.2001  18:00  4.2.2001  9:00  475.0
# ----------------------------------------
```

figure 29 file for the user data of the booked flights

5.3.2 Database Design

The data of the hotel provider are more complex. It is easier to handle it in a database. The different prices for the flight providers at the different locations and the connections between hotel and flight providers can be modelled in database relations.

The following script files show the SQL statements to build the database. The `locations` relation contains one data record for every location of a hotel provider with the total number of rooms and the price for one single room.

```
-- ---------- file  locations.sql ----------
--
-- master project
-- table for the locations of the hotels of one hotel provider
--

CREATE TABLE locations  (
       city           VARCHAR(40),
       totalrooms     NUMBER,
       price          VARCHAR(10),
       PRIMARY KEY    (city)
);
-- ----------------------------------------
```

figure 30 script to create the locations table

The `availability` relation delivers the available rooms of every day. This is the reason, why the table has a city, a date and a number column. If a user orders a

hotel room, the `available` value will be reduced by the quantity of ordered rooms. If the user orders a period of time, every day is handled separately.

```
-- ---------- file  availability.sql -------
--
-- table with the actual available rooms for every day

CREATE TABLE availability  (
       city           VARCHAR(40),
       hoteldate      DATE,
       available      number,
       PRIMARY KEY    (city, hoteldate)
);
-- ----------------------------------------
```

figure 31 script to create the availability table

The `flightproviderreduction` table is responsible for a special price, if there is a contract between the hotel provider and the flight provider who started the mobile agent.

```
-- ----- file  flightproviderreduction.sql -
--
-- table to calculate the price of a flight provider

CREATE TABLE flightproviderreduction  (
       flightprovider         VARCHAR(30),
       city                   VARCHAR(40),
       flightproviderprice    VARCHAR(10),
       PRIMARY KEY    (flightprovider, city)
);
-- ----------------------------------------
```

figure 32 script to create the flightproviderreduction table

If the requested number of rooms is available, the reduction in the `availability` relation is executed first and then the user information is written in the `booking` table. This relation contains all records of the users who ordered a hotel room. The hotel provider uses this information to generate the invoice for every customer.

```
-- ---------- file  booking.sql ------------
--
-- table for the info of the user who ordered a hotel room

CREATE TABLE booking  (
       firstname      VARCHAR(50),
       surname        VARCHAR(100),
       cardnumber     VARCHAR(50),
       startcity      VARCHAR(100),
       destinationcity        VARCHAR(100),
       outwarddate    VARCHAR(50),
       outwardtime    VARCHAR(50),
       returndate     VARCHAR(50),
       returntime     VARCHAR(50),
       price          long
);
-- ----------------------------------------
```

figure 33 script to create the booking table

The `dropAll.sql` file removes all four tables from the database.

```
-- ---------- file  dropAll.sql ------------
--
-- master project
-- remove all tables
--

DROP TABLE AVAILABILITY;
DROP TABLE BOOKING;
DROP TABLE FLIGHTPROVIDERREDUCTION;
DROP TABLE LOCATIONS;
-- ----------------------------------------
```

figure 34 script to remove all tables

6 Implementation

6.1 Agent Characteristic

The functionality of the agent system ‘Grasshopper’ is provided on the one hand by the platform itself, i.e. by the core agencies and region registries, and on the other hand by agents running within the agencies. Agents can access the functionality of the local agency, i.e. the agency in which they are currently running, by invoking the methods of their super classes `Service`, `StationaryAgent`, or `MobileAgent`. Agents are separated into two groups. A stationary agent can’t move and stays on the host where it was generated. A mobile agent has the functionality to migrate actively. There are three stationary agent classes in this application.

The `InquiryStationaryAgent` is responsible for the graphical user interface of the user. The `StationaryFlightAgent` offers the service to request a flight and the `StationaryHotelAgent` accesses a database to retrieve available hotel rooms. There are three classes that are derived from `MobileAgent`. One is the `FlightInquiryAgent` that starts the trip at the users home and comes back. Another one is the `HotelInquiryAgent` that migrates from hotel provider to hotel provider. There can be many hotel inquiry agents during one session. Every flight provider generates a new one if a hotel is requested. The `BookingAgent` is the last mobile agent and manages the booking of the offer.

Both kinds of agents have an `init()` and a `live()` method. The `init()` method is executed once when the agent is generated. It can be compared with a constructor. The `live()` method must be overridden by the implemented class. This method can be regarded as the heart of each agent, since it specifies its designated task.

An object of the class `AgentInfo` holds all attributes of an agent. This class comprises all relevant information about a specific agent, such as its identifier, location, and state. It can be used to have a unique Identification of an agent and it is the return value if an agent is generated with the `createAgent()` method of the interface `IAgentSystem`.

6.2 Agent Migration

This application consists of a minimum of three mobile and three stationary agents. Every flight provider will generate a mobile hotel agent if the user requests for a hotel offer.

Stationary agents can be treated as services. They will be started once and after that they wait for mobile agents.

The complete task of a 'Grasshopper' agent, i.e. all activities to be performed during its entire life time, have to be covered by the method `live()`. It is an abstract method provided by the super classes `StationaryAgent` and `MobileAgent`. Since this method is declared abstract within the super classes, it has to be overridden by subclasses in order to avoid compile time errors. The statements comprised by the method `live()` of a 'Grasshopper' mobile agent are separated into several execution blocks. Each execution block is completely executed within a single place on one location, and the last method of each block is the `move()` method, i.e. after the performance of an execution block the agent moves to the next agency.

After each migration, the agent has to determine which execution block to perform next. This is achieved by encapsulating the execution blocks into a `switch` statement or several `if` statements. The variable that indicates which execution block to perform is named `agentState`. After each migration, i.e. after each performance of the `move()` method, the `live()` method is executed from the beginning. However, depending on the current value of the `agentState` variable, only the execution block associated with this value is executed. The value of the state variable has to be incremented before each migration.

```
public class FlightInquiryAgent
{
    private int agentState;

    public void live()
    {
```

```
        switch( getAgentState() )
        {
            case 0:
                // state 0 is executed at the user's home
                :
                :
                // the state of an agent is used to execute a special fragment of this
                // switch statement
                setAgentState( 1 );
                try
                {
                    // agent migrates to the first hotel service offer
                    move( getAgentInfoFlightProvider()[0].getLocation() );
                }
                catch (Exception ve)
                {
                    System.err.println("Agent cannot move!");
                    ve.printStackTrace();
                }
            break;

            case 1:
                // the agent migrated to the first flight provider - he still has the
                // according data of the region and he can contact the stationary agent
                :
                :
                // check if the journey will go on
                if ( getCurrentStationNumber() < getAgentInfoFlightProvider().length )
                {
                    // there are some more flight providers
                    try
                    {
                        move( getAgentInfoFlightProvider() [getCurrentStationNumber()]
                            .getLocation() );
                    }
                    catch (Exception ve)
                    {
                        ve.printStackTrace();
                    }
                }
                // the agent visited all flight providers
                else
                {
                    // this was the last flight provider
                    // go back to the user who stated the agent and deliver the result
                    setAgentState( 3 );
                    try
                    {
                        move( getInfo().getHome() );
                    }
                    catch (Exception ve)
                    {
                        ve.printStackTrace();
                    }
                }
            break;

            case 3:
                // the agent visited all flight providers -
                // the collected information is passed to the stationary agent of the
                // user's home - the mobile agent is terminated
                :
                :
                // remove this flight inquiry mobile agent, (trip is ready)
                try
                {
                    remove();
                }
                catch( Exception ex) { ex.printStackTrace(); }
            break;
        }
    }
}
```

The `live()` method starts with state 0 to move to the first flight provider received from the `getAgentInfoFlightProvider()` method. The flight inquiry agent stays in state 1 until it has travelled to all flight providers. This is the reason, why `setAgentState( 3 )` is called only after all flight providers have been visited. If there are some providers left, the agent moves to these providers and state 1 will be executed again. In state 3 the agent is back on the user's home.

The `live()` method of a stationary agent is empty. Sometimes it contains only a system output line or it includes the start of the graphical user interface.

```
public class InquiryStationaryAgent
{
    public void live()
    {
        // show input dialog
        getQuiryDialog().setVisible( true );
    }
}
```

6.3 Agent System Information Requesting

When an agent starts its trip to travel through the network, it has to get the information of where to go to first. This information is retrieved in the first state of the `live()` method, that means in the agency of the user.

```
public class FlightInquiryAgent
{
:
:
  public void live()
  {
    case 0:
    // The agent is called for the first time and the according data
    // of the region has to be collected.
    // The agent is still located at the users home.
    // Check if there are flight providers and get how many there are.

    IRegion iRegion = getRegion();

    // Search filters are used to reduce the result set
    SearchFilter filter = new SearchFilter();

    // filter for the flight providers
    filter.setFilter( SearchFilter.NAME + "=StationaryFlightAgent" );
    setAgentInfoFlightProvider( iRegion.listStationaryAgents(null,filter) );

    // filter for the hotel providers
    filter.setFilter( SearchFilter.NAME + "=StationaryHotelAgent" );
    setAgentInfoHotelProvider( iRegion.listStationaryAgents(null,filter) );

    // if no flight provider is available, the mobile agent won't start -
    // no flight means no request possible
    if ( getAgentInfoFlightProvider().length == 0 )
    {
        try
        {
            remove();  // remove the mobile agent
        }
        catch( Exception ex) { ex.printStackTrace(); }
        break;
    }
```

```
    // if no hotel provider is found in the registry, it is not necessary to create
    // an hotel agent
    if (getAgentInfoHotelProvider().length == 0 )
    {
        // set flag
        setHotelProviderAvailable( false );
    }
    else
    {
        setHotelProviderAvailable( true );
    }
  }
}
```

All agencies and all their located agents are registered at a region. A region holds a registry, which can be asked for its agents, i.e. for mobile and stationary agents. The specific flight of hotel services can be received with a `SearchFilter`. If there aren't any stationary agents with a specific service, the flight inquiry agent will be deleted, or without a hotel service no mobile hotel agent is generated. This is indicated with a `hotelProviderAvailable` boolean flag. The method `listStationaryAgents` of the Region class sets the `agentInfoFlight-Provider` object to know where to go to in order to find a flight service.

6.4 Agent creation

This is how mobile agents can be generated. It is deployed to create the `InquiryFlightData`, the `InquiryHotelData` and the `BookingData`. The following code shows the creation of a mobile hotel agent initiated by the mobile flight agent. The same implementation is used of the `InquiryStationary-Agent` to create the `InquiryFlightData` and the `BookingAgent`.

```
if ( isHotelProviderAvailable() && inquiryFlightData.isHotelRequired())
{
    // generate an hotel inquiry agent -
    // the following values will be passed to the agent:
    // 1.  identifier of this mobile flight inquiry agent
    // 2.  info about the stationary agent that generated
    //     this flight agent
    // 3.  list of all hotel providers
    // 4.  flight provider on which host the hotel agent was
    //     started

    Object obj[] = {
        getInfo().getIdentifier(),
        getProducerInfo(),
        getAgentInfoHotelProvider(),
        flightProxy.getFlightProvider(),
    };

    try
    {
        String hias = "org.agent.flysystem.mobileagents.HotelInquiryAgent";
        AgentInfo ai = getAgentSystem().createAgent(hias, getProducerInfo().getCodebase(),
            getAgentSystem().getInfo().getLocation().getPlace(), obj);

        IHotelInquiryAgent inquiryAgentProxy = null;
        if (ai != null)
        {
            // collect all the unique agent identifer of all generated hotel agents
            this.getHotelInquiryAgents().add(ai.getIdentifier());
```

```
            inquiryAgentProxy = (IHotelInquiryAgent) ProxyGenerator.newInstance
                (IHotelInquiryAgent.class, ai.getIdentifier() );
            inquiryAgentProxy.setInquiryFlightData(inquiryFlightData);
        }
    }
    // catch special exception
    catch (AgentCreationFailedException acfe)
    {
        System.err.println("-- AgentCreationFailedException - Cannot create Agent --");
        acfe.printStackTrace();
    }
}
```

The `createAgent()` of the interface `IAgentSystem` takes four parameters. The first is the name of the class of the agent. The next are the location where unknown classes should be loaded from and the place where the agent should be created. The last parameter is a very important one. It is an array of instances of the class `Object`, which is passed to the `init()` method of the agent. These instances can be handled as start parameters of the agent. In the Flight Booking System the `obj` object consists of three `AgentInfo` objects and one String.

Another possibility of passing some values to an agent is shown after the agent creation. The `AgentInfo` object of the `createAgent()` method can be used to generate a new instance of the mobile hotel agent with the `newInstance()` method of `ProxyGenerator`. The object `inquiryAgentProxy` is an instance of `IHotelInquiryAgent` and can call all methods of the interface. Here it is `setInquiryFlightData()` that passes the `inquiryFlightData` to the object.

6.5 Agent communication

A great feature of the agent technology is the moving code and the possibility to connect to software on other machines. The first code fragment shows how to contact a stationary agent when the mobile agents are on their way to the network.

```
// proxy object for the communication with the flight provider
private transient IStationaryFlightAgent flightProxy;

// create proxy object of the stationary flight provider
flightProxy = (IStationaryFlightAgent)ProxyGenerator.newInstance(
   IStationaryFlightAgent.class, getAgentInfoFlightProvider()[getCurrentStationNumber()-
   1].getIdentifier() );

// start request for the flight information
Vector outwardVector = flightProxy.findFlights(inquiryFlightData);
```

When the mobile flight inquiry agent arrives at a flight provider, the `Proxy-Generator` generates an object of `StationaryFlightAgent` with the `AgentInfo`. The method `findFlights()` of this instance can be used to receive the vector of matching flight data. The member `flightProxy` has the

identifier `transient` because if the class migrates this member doesn't move with it.

Another communication between agents takes place when the mobile flight agent comes back to the user's home. The mobile hotel agent has to connect again to the stationary user agent, which generated it.

```
public class FlightInquiryAgent
{
:
:
  // getProducerInfo().getIdentifier() is the unique ID of the inquiry stationary agent,
  // which started this mobile agent -
  // invoke the 'action' method of the stationary agent and use the object
  // of this mobile agent to get the collected data
  try {
    agencyProxy.invokeAgentAction(getProducerInfo().getIdentifier());
  }
  catch (Exception e)
  {
    e.printStackTrace ();
  }
:
:
}
```

The flight inquiry agent knows, which user agent generated it. The agent info can be requested with the `getProducerInfo()` method and this data is enough to call the stationary user agent with the `invokeAgentAction()` method. The `action()` method of `InquiryStationaryAgent` is invoked and the collected flight data can be requested from the mobile agent with the `getFlightOffers()` method of `inquiryAgentProxy`. The returning gathered flight offers vector can be handled by the stationary agent.

```
public class InquiryStationaryAgent
{
:
    /**
     * If an instance of this class is invoked with the clone method,
     * this action method is called.
     */
    public void action()
    {
        // Object inquiryAgentProxy is the instance of FlightInquiryAgent,
        // which was sent out to collect the flight data.
        // Set the gathered flight offers in order to hold them in this stationary agent.
        if (getGatheredFlightOffers() != null)
            flightVector = null;
        getAnswerMap().clear();
        setGatheredFlightOffers (inquiryAgentProxy.getFlightOffers());
    }
:
}
```

At the end of the mobile hotel agent's journey it reaches the stationary user agent. The hotel agent was generated at a flight provider location and it appears the first time at the user's home. The agent generates a new instance with the `getProducerInfo()` method and calls the `setGatheredHotels()` method to pass the collected matching hotel data.

```
public class HotelInquiryAgent
{
:
[case 2 of the live() method]

            case 2:
                // agent visited all hotel providers
                // move the data to the inquiry stationary agent

                producerProxy = (IInquiryStationaryAgent)ProxyGenerator.newInstance(
                    IInquiryStationaryAgent.class, getProducerInfo().getIdentifier() );

                producerProxy.setGatheredHotels( this.getInfo(), getHotelVector() );

            break;
:
}
```

The `setGatheredHotels()` method of the `InquiryStationaryAgent` class is invoked by the mobile hotel agent. The passed values will be put in a map to have a relation between the hotel agent and the offer vector. The method has the identifier `synchronized`, because all generated hotel agents access this method to set the gathered information. Only one agent can use it at one time and there won't be confusion.

```
public class InquiryStationaryAgent
{
    /**
     * Method to add collected hotel information.
     * This method is synchronized, because every hotel agent uses this method to set
     * it's gathered data to this stationary agent.
     * Every generated hotel inquiry agent adds its information here.
     */

    public synchronized void setGatheredHotels(
        AgentInfo hotelAgent, Vector hotelOfferVector)
    {
        if (hotelAgent == null)
            return;

        // store the hotel agent identifier as the key for the hotel offer vector with the
        // data of the mobile hotel agent quiry
        this.getHotelMap().put( hotelAgent.getIdentifier(), hotelOfferVector );
    }
}
```

6.6 Algorithm to handle the Collected Data

A much more challenging task is to connect the gathered flight data with the corresponding hotel data. Both search results are delivered by different agents and at a different time. You can't predict which one comes first. When a hotel agent reaches the user's home, it calls the `setGatheredHotel()` method and puts the hotel agent identifier and the vector of offers in a bundle in the hotel map.

When a mobile flight agent arrives at the user's home again, it calls the `action()` method and assigns the vector of flight offers to the `flightVector` object in the stationary user agent. First the answer map is filled with the flight provider and its flight offer. The flight agent knows which hotel agents he has generated. The flight agent starts to check if the first hotel agent has reached the user's home. If the hotel agent hasn't arrived yet, the flight agent would wait for two seconds and check the situation again. If the hotel agent has arrived it can be found in the hotel map and the execution can continue. That means that a request answer object is generated for the matching flight data and the matching hotel data that is taken from the hotel map.

The first step is to remove old data, that is data of a previous request. That means to clear the flight vector and the answer map. A `FlightOffer` object contains the flight provider and the offer. This information is assigned to a new `Request-Answer` object that is also used to link the hotel data to this object. The second part of this process is the search for the hotel. The flight provider runs through the list of all hotel agents that it released during the trip through the network and waits if one mobile hotel agent hasn't returned to the user's home. When the hotel agent is there the map with the relation of 'flight provider' and 'request answers' will be checked and the hotel part is assigned to that `RequestAnswer` object linked to the flight provider in the map. After that a `RequestAnswer` object contains all flight offers of one provider and all hotel offers of all hotel providers, which were collected from the mobile hotel agent started at the flight provider.

```
public void action()

if (getGatheredFlightOffers() != null)
  flightVector = null;

getAnswerMap().clear();

setGatheredFlightOffers (inquiryAgentProxy.getFlightOffers());

// if the mobile agent didn't find any flights, a default unsuccess text appears
if (!getGatheredFlightOffers().isEmpty())
{
  // first :   the flight part of the result
  // the vector of gathered flights consists of FlightOffer objects
```

```
for (int go = 0; go < getGatheredFlightOffers().size(); go++)
{
  if (getGatheredFlightOffers().elementAt(go) instanceof FlightOffer)
  {
    FlightOffer fo = (FlightOffer)getGatheredFlightOffers().elementAt(go);
    // prepair a RequestAnswer object with the flight data and
    // put it in the answer map with the flight provider as the key
    getAnswerMap().put(fo.getProvider(), new RequestAnswer(fo));
  }
}

// second :   the hotel part of the result
// get the set of all released hotel agents of the flight agent
// the released hotel agents are distinguished by their identifier
HashSet flightHotelAgentSet = inquiryAgentProxy.getHotelInquiryAgents();

// If the hotel agent is part of the hotel map,
// that means that the hotel agent arrived at the user's home and
// the vector of matching hotel data will be assigned to the ReqestAnswer object
// belonging to the according flight provider.
// If the hotel agent is not part of the hotel map,
// that means that the hotel agent didn't arrived at the user's home until now and
// the stationary agent waits until it will arrive.

Iterator i = flightHotelAgentSet.iterator();

// walk through the list of all released hotel agents and
// find them in the hotel map
while (i.hasNext())
{
  // vector which is stored in the hash map for one hotel agent key
  Object vectorOfKey = null;
  // one entry of the hotel agent list belonging to the flight agent
  Identifier hotelAgentInfo = (Identifier)i.next();

  // endless loop, which is broken, if the agent identifier is found in the agent map,
  // that means, that the agent with this identifier is also back at the user's home

  while (true)
  {
  vectorOfKey = this.getHotelMap().get(hotelAgentInfo);
  // if there is no entry in the map for this identifier,
  // the control flow is stopped for two seconds and after that time
  // the loop will go on and invoke the getHotelMap() method again
  if (vectorOfKey != null)
  {
    // the loop can be stopped because the hotel agent identifier is found
    break;
  }
  else
  {
    // break of the contol flow
    // the assumption is, that a hotel agent arrives and sets a new entry in the hotel
    // map
    System.out.println("---- I wait for 2 seconds");
    try
    {
      Thread.sleep(2000);
    }
    catch (Exception e) { e.printStackTrace(); }
  } // end of the endless loop

  // the entry for the requested agent identifier is found in the map
  if (vectorOfKey instanceof Vector)
  {
    // vector of matching hotel data
    Vector vectorMHD = (Vector)vectorOfKey;


    // get the flight provider

    if (!vectorMHD.isEmpty())
    {
      MatchingHotelData mhd = (MatchingHotelData)vectorMHD.get(0);
```

```
        RequestAnswer ra = (RequestAnswer)getAnswerMap().get(mhd.getFlightProvider());

        // set the hotel part of the RequestAnswer object
        // belonging to the request of the same flight provider
        ra.setHotelPart(vectorMHD);
      }
    }
  }
}
```

6.7 Accessing the file

The `lineReading()` method is called with the path and the file name of flightdata.dat and reads one line after the other. The method `evaluateLine()` takes a line as a String and generates a `MatchingFlightData` object to be able to compare it with the request data `InquiryFlightData`. The `checkAvailability()` method accesses the availability.dat file to check the actual data. The last action is to close the `FileReader` and the `LineNumberReader` objects.

```
    /**
     * method to read the flight data from the file
     */
    private void lineReading(String absoluteFile)
    {
        LineNumberReader reader;

        // variable for a temporary storage of a line
        String str;

        // line number of the current line
        int lineNumber = 0;

        File inputFile = new File(absoluteFile);
        try
        {
            FileReader in = new FileReader(inputFile);
            reader = new LineNumberReader(in);

            // repeat until all lines are read
            while ((str = reader.readLine()) != null)
            {
                lineNumber = lineNumber + 1;
                try
                {
                    if ( str.charAt(0) == ' ' || str.charAt(0) == '#')
                    {
                        // a line starting with a blank or with #
                        // will be ignored
                        continue;
                    }
                }

                catch (IndexOutOfBoundsException ioob)
                {
                    // If a line has no entry, charAt() throws a
                    // IndexOutOfBoundsException
                    // (line will be skipped)
                    continue;
                }
                // disjoint a line
                try
                {
```

```
                    // evaluateLine returns a MatchingFlightData object
                    data = this.evaluateLine( str );

                    checkAvailability(data);
                    // add to the vector; this is only executed if no exception occurs
                    dataList.add(data);
                }
                catch ( ProviderFileCoruptException ex )
                {
                    // If a line is not correct, an error is shown on the command line.
                    // The execution of the file will go on.
                    log("DATA FILE CORUPT !!!\n\tIn line number "+ lineNumber +
                        " ( Line ignored! )");
                }
            } // end of while

            in.close();
            reader.close();
        }
        catch(IOException e)
        {
            System.err.println(e.getMessage());
            e.printStackTrace();
        }
    }
```

6.8 Accessing the database

The database model includes four tables. After the connection and the login are successful, the `Location` relation is checked for the location and the total number of rooms and the standard price is fetched. The second SQL statement checks the `Availability` relation for every requested day. The smallest number of available rooms in the period of time is the number of available rooms to look for. The `FlightProviderReduction` relation models the contracts between the flight and the hotel providers. If a hotel chain has special prices for a flight provider in a special city, the flight provider price can be retrieved from this table and it is replaced with the standard price.

```
public synchronized MatchingHotelData matchedHotel(InquiryFlightData hotelData)
{
    // create one object to fill it with data and send it back
    MatchingHotelData  mhd = new MatchingHotelData();

    // set the number of available rooms on a not reachable value
    // (this is important to get the minimum amount of rooms)



    int availableRoomsDefault = 9999;
    int availableRooms = availableRoomsDefault;
    int totalRooms = 0;
    String standardPrice = null;
    String fpPrice = null;
    Connection con = null;
    Statement stmt = null;

     try {
        Class.forName("oracle.jdbc.driver.OracleDriver");
```

```
        String host = dbMessage.getString("dbhost");
        String port = dbMessage.getString("dbport");
        String databaseName = dbMessage.getString("dbname");
        String user = dbMessage.getString("dbuser");
        String password = dbMessage.getString("dbpw");

        con = DriverManager.getConnection("jdbc:oracle:thin:@" + host + ":" + port +
              ":" + databaseName, user, password);
        stmt = con.createStatement();

        // first SQL statement to contect the locations table
        // get the number of rooms and the standard price of one room
        ResultSet rs = stmt.executeQuery( "select totalrooms, price from locations
                       where " + "city = '"+hotelData.getTargetAirport()+"'");
        while (rs.next())
        {
            totalRooms = rs.getInt("totalrooms");
            standardPrice = rs.getString("price");
        }
    } catch (Throwable e) {
        System.out.println("error in statement for table locations");
        e.printStackTrace();
    }

    try {
        :
        :
        // second SQL statement to contect the availability table

        // reduce one day, because the last day isn't important
        //endDate.add(GregorianCalendar.DAY_OF_MONTH, -1);
        //end.setDate(endDate);

        // The start date is iterated, until the start date is equal to the end date.
        // The table availability holds for every day the amount of free rooms.
        while (!(start.sameDate(end)))
        {
            // the statement (startDate.get(GregorianCalendar.MONTH)+1) is necessary,
            // because the month is zero based
            ResultSet rs = stmt.executeQuery("select available from availability where
              city = '"+hotelData.getTargetAirport()+"' and hoteldate = TO_DATE" +
              "('"+startDate.get(GregorianCalendar.DAY_OF_MONTH)+" -
              "+(startDate.get(GregorianCalendar.MONTH)+1)+"-" +
              startDate.get(GregorianCalendar.YEAR)+"','dd-mm-yyyy') ");

            int a = 0;
            while (rs.next())
            {
                a = rs.getInt("available");
            }
            //
            if (a < availableRooms && a != 0)
            {
                availableRooms = a;
            }

            // the value of the startDate object is increased by one day
            startDate.add(GregorianCalendar.DAY_OF_MONTH, 1);
            // this change is set to the start object
            start.setDate(startDate);
        }
    }
    catch (Throwable e)
    {
        e.printStackTrace();
    }

    // If the variable availableRooms has still the value of availableRoomsDefault,
    // that means that, there aren't any rooms booked and all rooms are available.
    // (the amount of available rooms is the total number of rooms)
    if (availableRooms == availableRoomsDefault)
        availableRooms = totalRooms;
    try {
        // third SQL statement to contect the flightproviderreduction table
```

```
        ResultSet rs = stmt.executeQuery( "select flightproviderprice from
             flightproviderreduction where " +"flightprovider = '"+flightProviderName+"'
             and city = '"+hotelData.getTargetAirport()+"'");
        while (rs.next())
        {
            fpPrice = rs.getString("flightproviderprice");
        }
    }
     catch (Throwable e)
        {
            System.out.println("error in statement for table flightproviderreduction");
            e.printStackTrace();
        }
        :
        :
        return mhd;
    }
```

The first SQL statement looks for the standard price and the total number or rooms for on hotel. The second SQL statement checks the availability. Every day has an entry in the `Availability` table with the actual free rooms for that day. The algorithm is to begin at the start date and check all days until the end date of the request is reached. The smallest amount of rooms is the available number for the requested period. The third SQL statement looks for a special price for a flight provider.

7 Conclusions and Further Work

7.1 Other Flight Booking Systems

This kind of application combines two actions implicitly. It searches for currently available flights and hotel rooms. Without the Flight Booking System (FBS) of this project the hotel and flight requests have to be executed sequentially one after the other.

Some other providers like '*www.flights.com*', '*www.travel24.com*' or '*www.flug.de*' with similar services want to know the name and address of the users before they begin the request. This has nothing to do with the booking. The FBS just wants to know the user information if the user books an offer and not to get some offers.

The investigation has exposed that there are some providers with flight, hotel and car hiring services, but all offer three independent tasks. The user has to enter the data three times and all requests are stand-alone. No provider has a combination of the services and a single user data inquiry like the FBS.

The services of '*www.eflug.de*' can be compared with these of the FBS. They are airline independent. The disadvantage is, that after the first request the user didn't know anything about the availability. After a second request, the user knows if it is possible to book a flight or not. After the search the user of '*www.giata-hotel-guide.de*' gets a hotel chain independent result set of rooms to book, but only after an e-mail reply he or she knows if the room is still possible to book or not. This reply can reach the user some hours later.

A benefit of the Flight Booking System of this project is, that it only displays a combination of available flights and hotel rooms. All services on the Internet list flights or hotels but not both together and the offers are only the possibilities and there is no predication of the disposability. Some providers offer a second request to know if the flight or hotel room is available or not. Other providers like '*www.expedia.de*' only brief the customer about available flights after they booked it. This isn't satisfying because sometimes the user just wants to know if the flight is available without booking it. Every single step is executed separately.

Airlines offer their own flights at their Internet portal. The services of '*www.itn.net*' or '*www.travelnow.com*' compare the bargain of different providers. The FBS is a provider independent application too and considers all airlines or hotel chains which are registered at the region registry.

7.2 Conclusions

As shown in this dissertation it is possible to develop an application to book flights or hotel rooms with the use of mobile agents. These flight or hotel services can be distributed on the Internet. The aim was to analyse, design and implement the application, which was completely achieved. The mobile agent technology has advantages in the field of flexibility, network traffic, communication, modularity, and actuality compared to the Client/Server paradigm[5].

The developed application offers the user the possibility to find the cheapest offer of a flight or a flight and hotel combination by the time of the request. The tasks are requesting, comparing and booking an offer. The user can rely on the actuality of the data because the number of available seats and the price are checked with every request. The great benefits are that the result is independent of the provider, all offered flights and hotel rooms are regarded, no information will be skipped, the cheapest offer can be chosen by the user with a single click and the booking is very easy.

One possibility to design an application like this is to use mobile agents. These autonomous software agents travel to a network and visit special hosts. An agent system has to run on the participating locations. Chapter 2 ‘A Survey of Mobile Agent Systems’ describes three different systems and compares them. The result of the comparison was, that the Grasshopper agent system is best qualified for this project.

This application has been realized with the approach of the software lifecycle. First the requirements were established and the use cases were designed. After that the scenarios helped to find the subjects and the build classes with the attributes. During the design phase all hardware and software constraints were prescribe and the persistence model for the file and database storage was designed.

7.3 Further Work

After the application is started, a graphical user interface is opened to enter the request data. Every user who wants to use the application starts it on its local machine. One recommendation for a further work is to design a new approach of the human computer interface. The service providers can be distributed on the Internet. This can also be adapted to the user interface.

[5] Interim Report section 2.3.3 Client/Server versus Mobile Agents

In order to provide the Flight Booking System to a greater quantity of users it can be used as part of a web page. Everyone can access it from all over the world. The big advantage is that the functionality and the data model can be over taken without any modifications. Only the front-end has to be changed. The first suggestion is to build an applet of the application and integrate it in a web page. The disadvantage is that the applet is loaded to the user and the execution is on the users machine.

Another suggestion is to design a servlet offering the user interface. The great improvements are the execution on a web server, the unrestricted access with a web browser and the fact that the user doesn't have to install the agent system on its computer. Servlets are pieces of Java source code that add functionality to a web server in a manner similar to the way applets add functionality to a browser. Servlets are designed to support a request/response computing model that is commonly used in web servers. In a request/response model, a client sends a request message to a server and the server responds by sending back a reply message.

This architecture can be applied in the case that the user has a form on a web page to enter its flight data and sends this data to the servlet. HTML can provide a rich presentation of information because of its flexibility and the range of content that it can support. Servlets can play a role in creating HTML content. Complex web sites often need to provide HTML pages that are tailored for each visitor, or even for each hit. Servlets can be written to process HTML pages and customize them as they are sent to a client.

The agents will be started on the web server and return to that. The functionality of the mobile and stationary agents the movement and the data model is the same like it is designed in this project. After all agents are back on the web server and the result is generated to display a HTML document or a java server page (JSP) can be generated individually of the search result. A JSP page is a text-based document that describes how to process a request to create a response. The description intermixes template data with some dynamic actions and leverages on the Java Platform.

8 Management of the Project

This project has been realized next to a normal job of an engineer. All phases of the software lifecycle were realized. There were five milestones defined to submit the documentation and the work that was done until that time. At the end of every milestone the documents and deliverables were sent to the supervisor.

Section 3 ‘Time-Plan’ of the Interim Report includes the detailed description of the project management and the development process. The available time was devided into four phases. It started at the backround phase, followed by the requirements, analysis, design, and implementation phase.

The time table is attached at the end of the Interim Report and the development and project management is explained in section 3.1.

9 Appendix

This section shows the application in use.

The graphical user interface of figure 35 shows the inquiry panel, which is the start dialog. The user enters its values of the date, the departure and destination city, the quantity of flights, and decides whether he or she wants a return flight or a hotel.

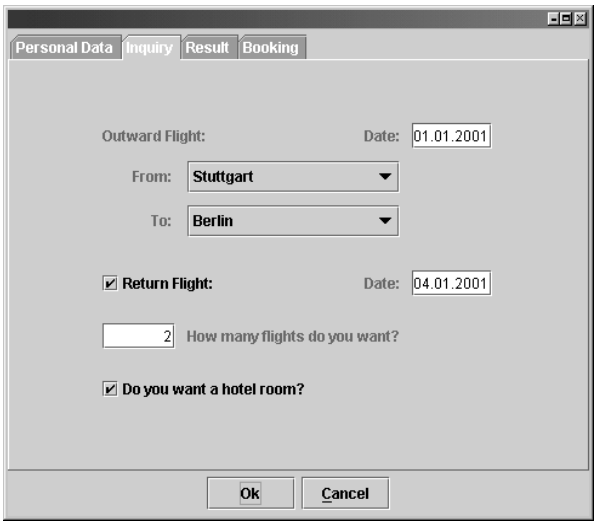


figure 35

After all generated agents returned to the user, the result window of figure 36 is opened. The user gets all possible information about an offer. He or she sees the available flights of the requested day. Offer 1 of this example (British Airways) consists of three flights for the outward way, one flight for the return direction and two hotels. The user perceives the available seats and the price of every flight. The same is valid for the hotel offers. The application calculates the price range of offer 1.

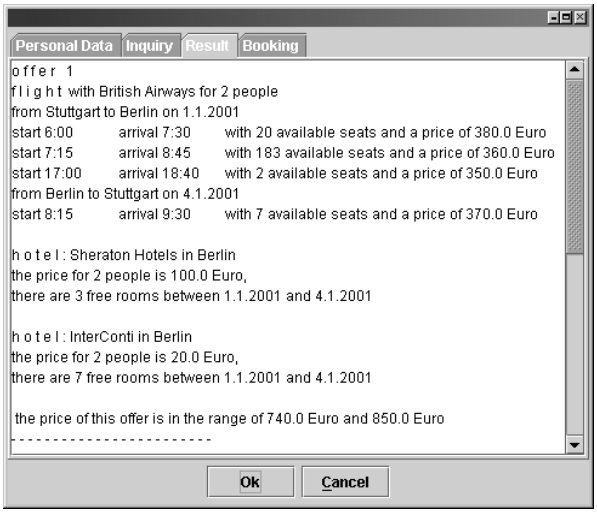


figure 36

Offer 2 is of another flight provider (TWA). It consists of two outward and two return flights. The available rooms of the hotels are equal to offer 1 but the prices differ because of the other flight provider and the contract between the hotel chain and the airline.

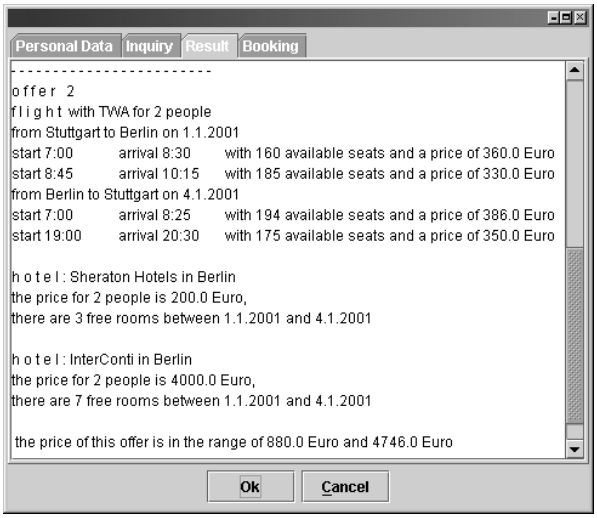


figure 37

Before the user can book an offer, he or she has to enter his or her personal data. The payment is organized with the credit card number.

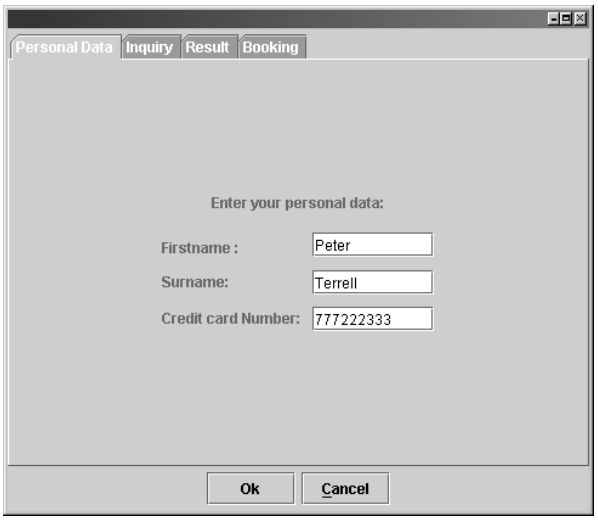


figure 38

Figure 39 shows the booking panel. The easiest way to book the cheapest offer is to mark the first opportunity of this page. The application has calculated the cheapest offer and the user can rely on this information without checking all possible offers.

If the user doesn't want to take the cheapest offer because of the departure time, he or she can enter the offer number of the previous panel (the offer panel). If there is more than one outward or return flight, the user has to choose one from the combo box like it is shown in figure 39.

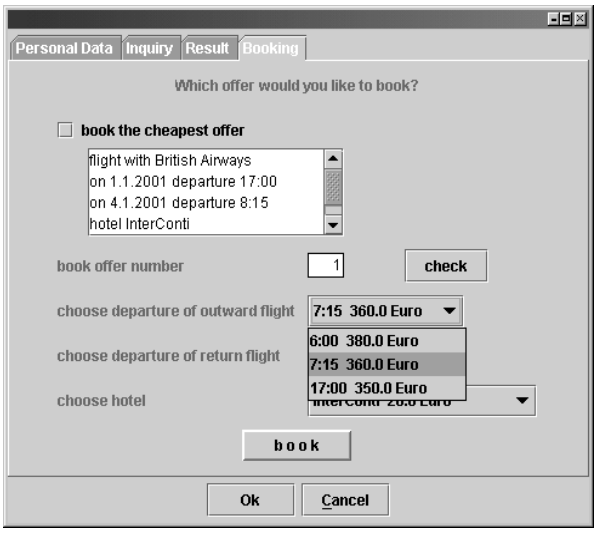


figure 39

After the user has defined its order with the specific departure time and the hotel chain, the booking can begin. The 'check' button fills the combo boxes if there is more than one possibility for the flights or the hotel room. Figure 40 shows a panel with the entire data to book an offer.

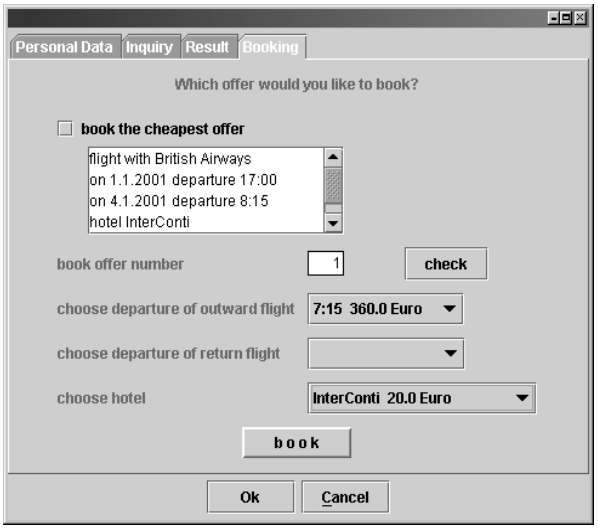


figure 40

If the booking was successful, the notification is shown at the user's home.

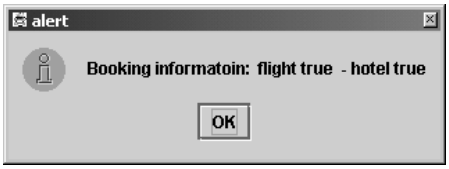


figure 41

The following figures show the trips of the mobile agents and the locations of the stationary agents. There are two flight and two hotel providers. The registry of figure 42 holds all providers offering a service. All flight inquiry agents request the service providers there.

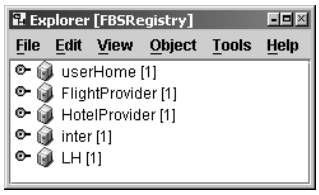


figure 42

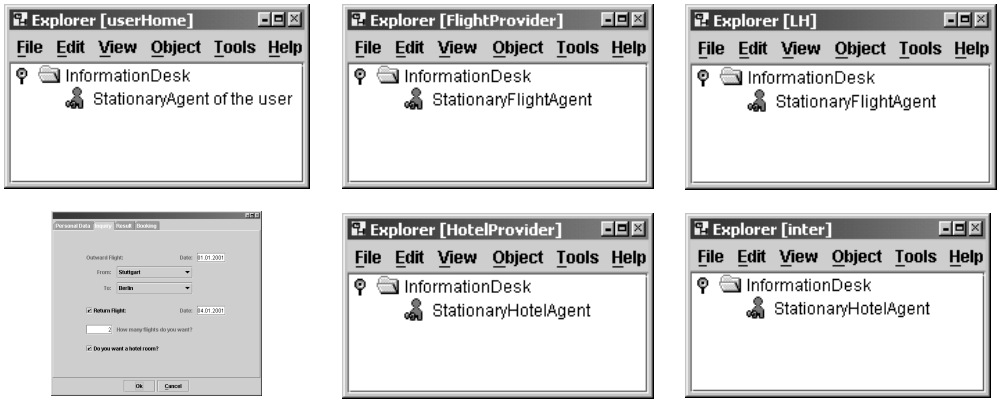


figure 43

After the user has started the Flight Booking System application, it launches the stationary agent of the user and opens a GUI[6]. The user can enter its data. The stationary agents of the service providers wait until a mobile agent arrives and communicates with them. After a provider has started its service, it is added to the registry of figure 42. The flight providers are called 'FlightProvider' and 'LH' and the hotel providers are called 'HotelProvider' and 'inter'. A flight inquiry mobile agent is generated at the 'userHome' to travel throw the network (figure 44).

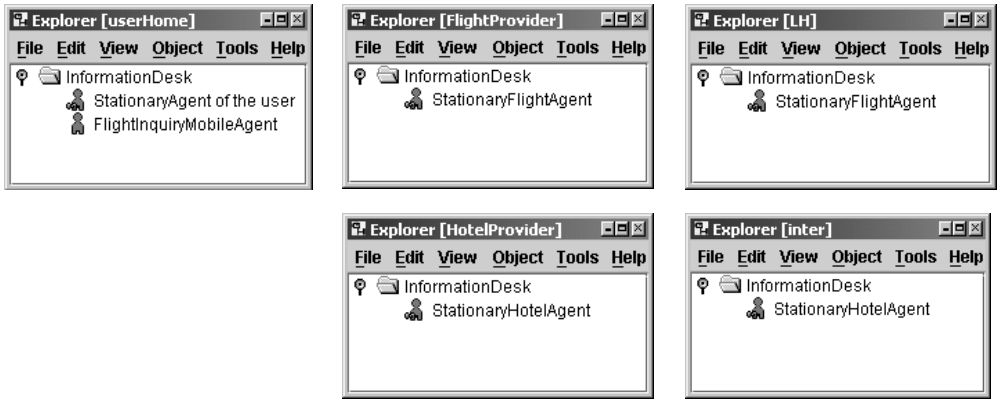


figure 44

[6] figure 35

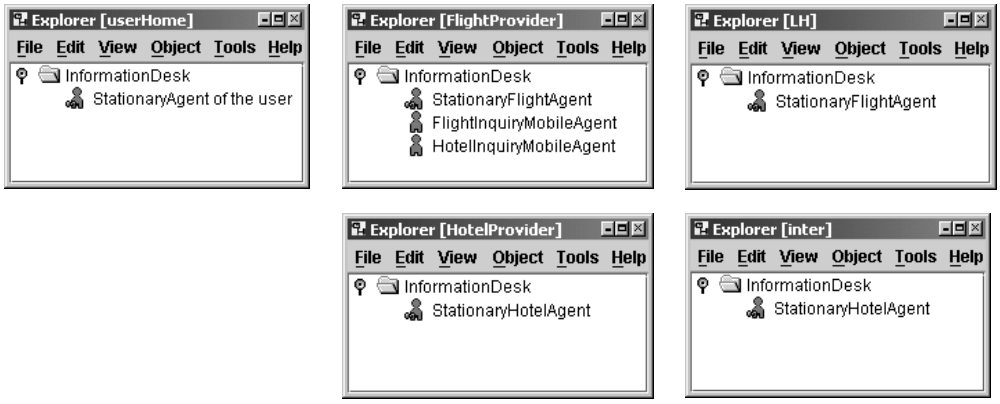


figure 45

The flight inquiry mobile agent moves to the first flight provider and contacts the stationary flight agent. The search for an available flight was successful and the flight agent generates a hotel inquiry mobile agent. The hotel agent has a list with all hotel providers and it starts its trip with the location ‘HotelProvider’. The flight inquiry mobile agent migrates to the location ‘LH’.

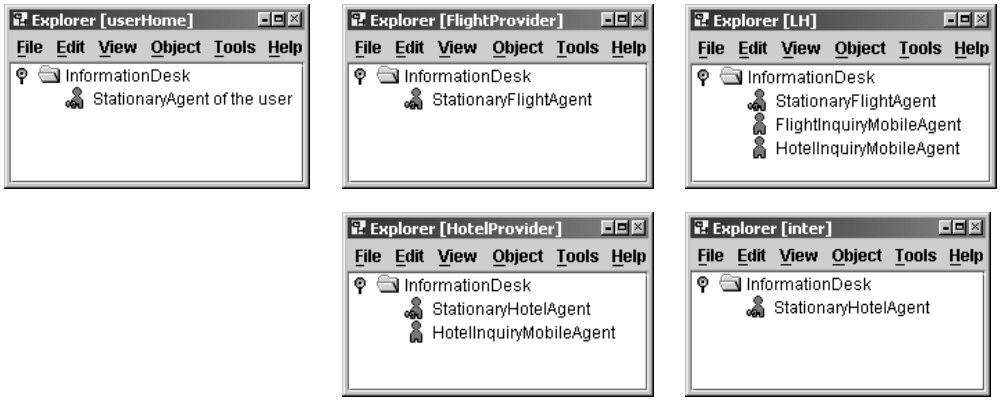


figure 46

The search for available flights at ‘LH’ was successful and another hotel inquiry mobile agent is generated (figure 46). It starts its trip at the location ‘inter’.

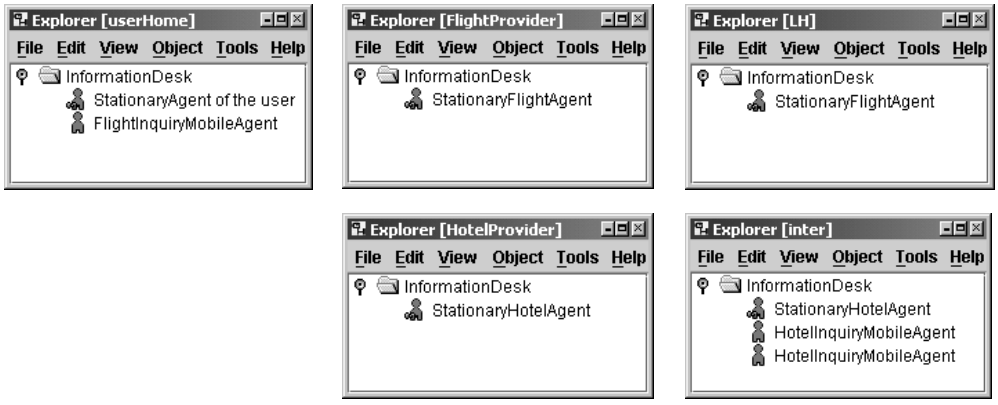


figure 47

The flight agent visited all flight providers and returned to the user’s home. The hotel agent, which was generated at ‘FlightProvider’, moves to the hotel ‘inter’.

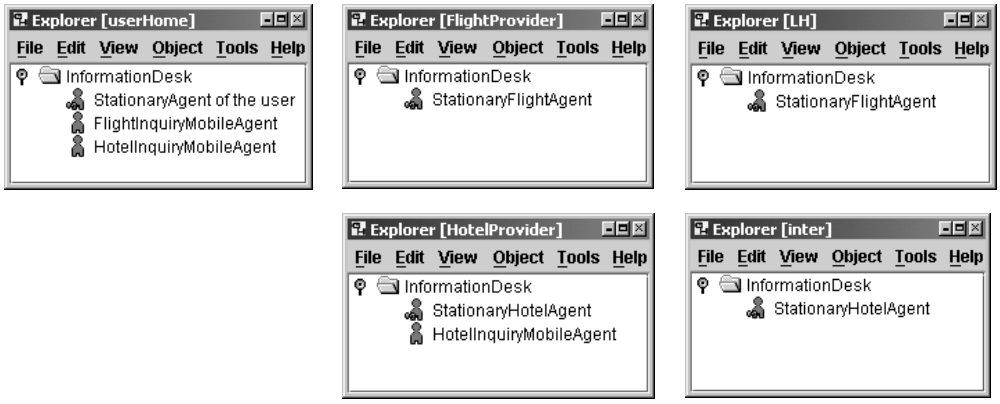


figure 48

The hotel agent, launched at ‘FlightProvider’, has finished all locations of the itinerary and it returns to the user’s home. The flight agent checks all incoming hotel agents whether it belongs to the flight agent. If there is a relation, a new request answer object is generated. The hotel agent of ‘LH’ moves from ‘inter’ to ‘HotelProvider’.

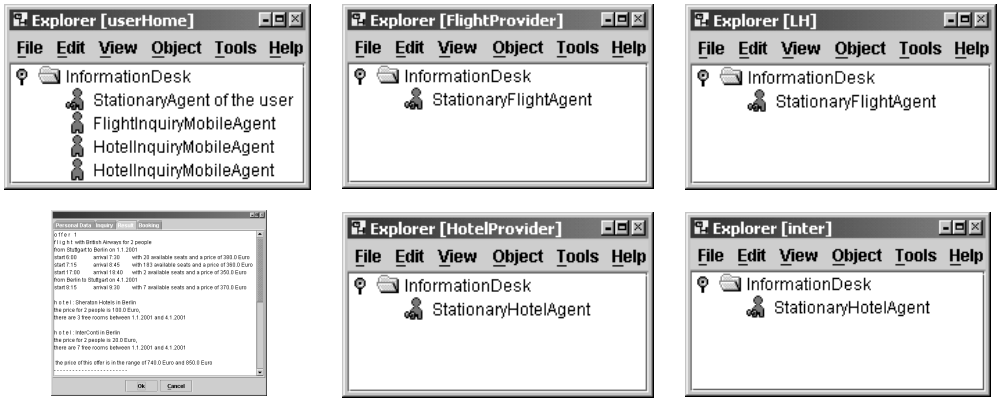


figure 49

The flight agent and both hotel agents have arrived at the user’s home and have assigned their data to the request answer. The result is displayed in the GUI.

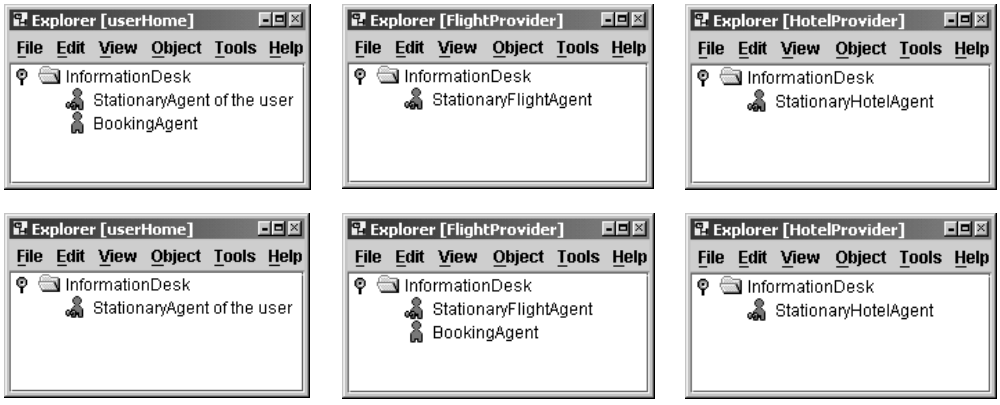


figure 50

Figure 50 shows the first two steps of the booking process. The BookingAgent starts at the user’s home and moves to the flight provider of the chosen offer.

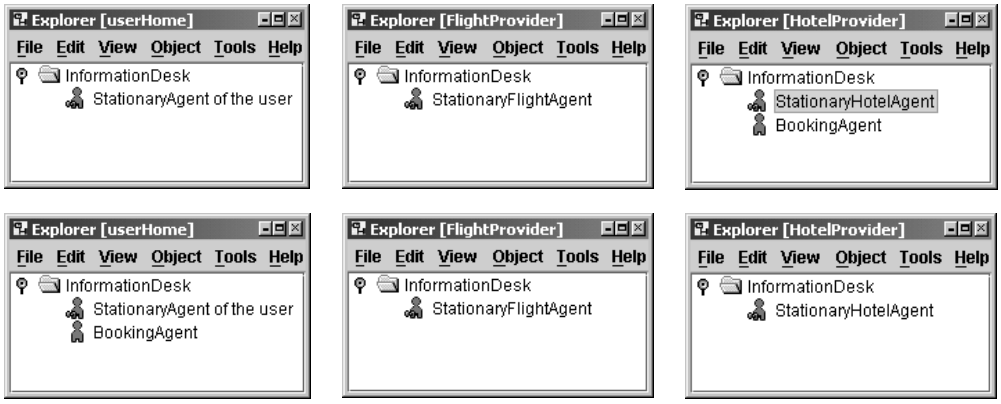


figure 51

The booking agent checks the availability of the requested number of flights and transfers the personal data of the user. Then it moves to the hotel provider and does the same. The second step of figure 51 shows the returned BookingAgent. After the user received the confirmation of the order he or she can close the application and the stationary agent of the user will be deleted.

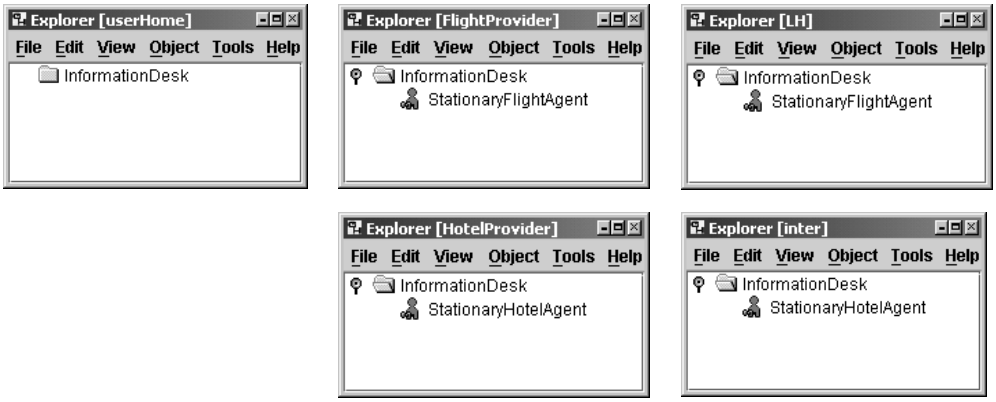


figure 52

The stationary agents of the providers are still alive and wait for other mobile agents. Figure 53 shows the booking result from the perspective of the providers. The personal data of the user is stored in the file `booking.dat` in case of the flight provider and in the database relation `Booking` in case of the hotel provider.

figure 53

10 References

The material listed in the References section is the quoted material. Material not referenced in the report is listed in the Bibliography section.

References

[Com00] Sun Microsystems, CeBit fair handout : How to .com – your business, 2000

[Goll97] Joachim Goll, Lecture notes of "Methods for Software Development", University of Esslingen, 1997

[GoRa97] Joachim Goll, Stephan Rath, Lecture notes of "Object Oriented Development", University of Esslingen, 1997

[Hau97] Dirk Hauber, object oriented analysis with UML, Kölsch&Altmann, 1997

[Hau98] Dirk Hauber, object oriented design and UML, Kölsch&Altmann, 1998

[Hru99] Peter Hruschka, object oriented development of real time systems, Software Technologie Beratung, 1999

[ikvPG98] Jan Mauersberger, ikv Grasshopper Development System, Programmer's Guide, 1998

[Karm98] Ahmed Karmouch, Mobile Software Agents: An Overview, IEEE Communications Magazine, 1998

[Kotz99] David F. Kotz, Robert S. Gray, Mobile Agents and the Future of the Internet, 1999

[Lan96] Danny B. Lange and Daniel T. Chang, IBM Aglets Workbench - A White Paper, 1996

[Lan98] Danny B. Lange, Mitsuru Oshima, Programming and Deploying Java Mobile Agents with Aglets, 1998

[Obj98] ObjectSpace, Voyager Core Technology 2.0, 1998

[Pro99] Roger W. Prowse, Lecture notes of "Software Engineering", Brunel University, 1999

[Scha99] Stephen R. Schach, Classical and Object-Oriented Software Engineering with UML and Java, McGraw-Hill, 1999

[Sun00] Sun Microsystems, CeBit fair handout : E-Commerce essentials, 2000

Bibliography

Dejan Milojicic, Trend Wars - Mobile Agent Applications, IEEE Concurreny, 1999

K. Rothermel and R.Popescu-Zeletin, Mobile Agents, Lecture Note in Comp. Sci. Series, 1997

The Agent Home Page, www.agent.org

Aglets Software Development Kit Home, www.trl.ibm.com/aglets

Grasshopper - the Agent Development Platform
www.grasshopper.de

Voyager Product Line,
www.objectspace.com/products/voyager

Overview of Mobile Agents,
agent.cs.dartmouth.edu/general/overview.html

Network of Agent-Based Computing, www.agentlink.org

UMBC AgentWeb Articles, agents.umbc.edu

Massachusetts Institute of Technology, Software Agents group
Media Laboratory, agents.www.media.mit.edu/groups/agents

Java Homepage, java.sun.com

Department of Electronic and Computer Engineering

MSc in Distributed Computing Systems Engineering
The Esslingen Assignment
1999 - 2000

Interim Report

Design and Implementation of a Flight Booking System on the Internet supported by Java-based Mobile Agents

Axel S Hallwachs

Supervisor: Dr M J D Wilson

3rd July 2000

Interim Report

Table of Contents

1 Introduction

One challenge of an engineer is the merging of different aspects of his or her profession in one project. In this project the programming of mobile agents will be joined with the Internet. The topic 'Mobile Agents' is becoming more and more important during the last years. This introduction introduces some key aspects to understand the relationship between the major tasks.

1.1 The Internet

Simply put, the Internet is a large set of computer networks that communicate with each other, often over telephone lines. It enables companies, organizations, individuals, schools and governments to share information across the world.

The Internet consists of eight parts:

- World Wide Web, viewing hypertext
- electronic mail, sending messages
- newsgroups, discussions about science, economy, profession, etc.
- chat, online marketplaces to meet people
- FTP, file transfer from and to remote computers
- telnet, work on remote machines, terminal emulation
- Archie, database search engine
- Gopher, central directory for different topic on the Internet

The Internet includes the **World Wide Web**, which enables you to see documents in richly formatted text and pictures. Many web pages link to other web pages, so it's easy to browse, a large amount of information by just clicking and navigating with the mouse. Most people talk about the Internet and think of the World Wide Web. This part would also be the part this project will be involved with.

The second part of the Internet is **electronic mail**. It helps you to speed up the contacts in business, to friends and to your family. You can communicate with them in seconds by using e-mail. You can send and receive messages immediately all around the world, and you don't have to look for an envelope or stamp. A **newsgroup** is a very important establishment for discussions about a specific topic. Every participant can tell his or her experience about that topic. You can get information about their problems working in the same area like you do. To have an electronic

conversation in “real time” with someone on the other side of the world, you can **chat** with him. With **telnet** you can access remote machines and work on them and use their calculation power. Files can be transferred with **FTP**.

Perhaps you are wondering how the Internet might help you. Do you want to plan a trip? Check out sports scores? Shop online books, clothes or even cars? Read online newspapers and magazines from around the world? All this is on the Internet. If you can’t find what you are looking for, you can search for all kinds of information by using search services. [Com00]

1.2 Motivation

The Internet is more and more involved in our daily life. It is as important as the telephone. The most important parts are the World Wide Web, e-mail, newsgroups and file transfer. The World Wide Web provides a graphical navigation through millions and millions of computers with web servers running on it. From a technical point of view, we can say that the Internet is a huge and open network of connected computers of different hardware, software, operating systems, etc. which can be accessed by everyone. But if we think of the Internet with an economical and commercial approach it is much more than this. Global communication takes us to a global village in which we can trade with everyone.

More and more activities can be dealt with the help of the latest web technologies. E-commerce is a great topic to manage things like ordering, shopping, online banking, stock trading, auctions, software downloading, message sending and infinitely more things. The Internet offers nowadays much more than just ‘ trans-mission of information’ like in the earlier days. To access all the information you like is very useful and enjoyable. All people have the same access to the same sources independently of where they are. This is great – but the Internet is much more powerful than this.

There is an economical substance in the Internet. Conceivable possibilities are initiations like online orders practised by bookshops or music stores. The customer can choose products of every vendor independently of time and date all around the world. The sellers can offer their products to a much greater audience. E-commerce opens the efficiency of supply chains and can reduce costs. It changes the relationship of businesses and customers. Companies’ organisations will be changed because of competition advantages. The electronic exchange

influences every enterprise, the logistic, the support and the purchasing department. You can develop a supply channel, which is never known.

If you provide goods, information and services with a secure Internet technology you talk about E-commerce. These are transactions to build a whole workflow on the Internet. Price arrangements can be created much faster as a reaction on the supply and demand than on traditional ways. The solutions are supply oriented and demand controlled.

Finance applications and enterprise resource planning are main topics. An electronic data interchange system manages the automatic obtaining of new material or a dynamic trade can be established from the raw material to the end user. [Sun00]

1.3 The Topic

Throughout the project selection process, the topic of my MSc dissertation crystallized as follows:

Investigations, design and implementation of a flight booking system on the Internet with an object oriented approach supported by mobile agents running on a Java-based agent system.

It meets the necessary attributes of a project, because of the distribution of the software over many computers we have a ‘distributed systems architecture’ and because of the software engineering aspects it is clearly relevant to the subjects of the course ‘Distributed Computing Systems Engineering’. The topic is an area that has a great potential in the future. It’s developments get more and more important because of the flexibility and it is personally interesting for me. It gives the opportunity to demonstrate a range of intellectual and applied skills.

1.4 Description

Nowadays a lot of orders are dealt via the Internet. Everyone who wants to get information about offered flights has to check all the travel agencies independently one after another. You go from one web page to the next and enter the same information with every single offer. If anyone doesn't know where to look, he or she has to search for all the travel offices first, to compare the prices for a special flight. Perhaps he or she doesn't know how to find the offices or where to find all the different services or supplies.

In conjunction with booking a flight, you often have the request for booking a hotel in the town you travel to. To get a free hotel room, you do the same actions as you did for booking a flight. That means, that you look for offers providing hotel rooms and you search the web in a time intensive action to compare the search results to get a cheap and nice opportunity. But you don't know if there is one you didn't recognize.

This is a great possibility for mobile agents, because you can construct a logic, which can be used again and again. Another reason for using mobile agents is the fact that the action could be inversed. The customer would not have to search for the offers, but the sellers would have to provide their services.

In the previous scenario, the user has to search explicitly and in detail for all offers he or she can find and it is always conceivable that he or she misses some opportunities.

In the approach of this project, the sellers carry out the first action and offer their products. They do this in a computer network. One part of the network is located with them and one central station has a registry with all offered services combined with the vendors.

The user who wants to book a flight and a hotel room just has to start a mobile agent with the necessary information of the date and the cities and the agent can collect all resources which are reachable. The user can rely on the result that all available offers are considered.

1.5 Features

The application of this project has the following basic features:

1. It is a flight booking system, which discovers a network for available services, namely flights and hotel rooms.

2. The user browses a special web page with a graphical user interface on it. He or she enters the flight information (the date and the start and destination locations). He or she starts a mobile agent to discover the resources.

3. Flight sellers, for example travel offices and hotels provide their services from their local hosts stored in a centralized registry. The mobile agent can retrieve this repository.

4. The travel offices have a contract with the hotels about the prices of the rooms. That means that different travel agencies have different prices for the same hotels.

5. A middleware has to run on every machine that is involved, because for the communication and the connection. The first task of the dissertation would be a query about different agent systems. The first topic is a comparison and then one of them will be chosen.

2 Background to the Project

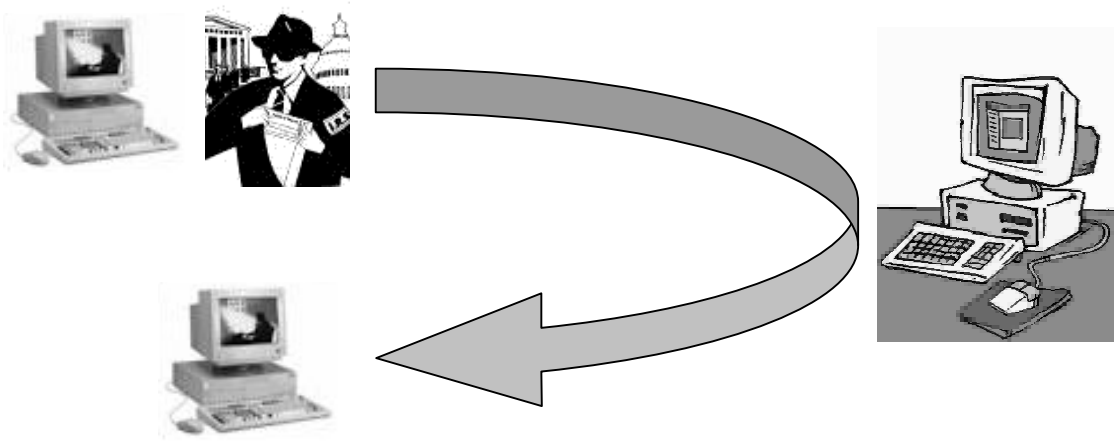
Current technological trends may lead substantially to a system based on mobile code, and in many cases – based on mobile agents. It seems, within a few years, nearly all general Internet sides will be capable of hosting and willing to host some form of mobile code or mobile agents.

2.1 Essentials of the Topic

2.1.1 Mobile Agents

Software agents are programmes that assist people and act on their behalf. Agents function by allowing people to delegate work to them that can be executed

autonomously and asynchronously. This is a very interesting concept that becomes even more attractive when the agents are no longer bound to the system where they begin execution. A mobile agent has the unique ability to transport itself from one system in a network to another. The ability to travel allows mobile agents to move to a system that contains services with which they want to interact and then to take advantage of being in the same host or network as the service.



The figure shows that an agent has a start location and moves to all different kinds of computer. During the itinerary all operating systems and all hardware components are possible. [Lan98] talks about a heterogeneous environment such as the Internet.

The ideas of mobile abstractions are probably as old as distributed systems. **Mobile agents** represent the next leap forward in the evolution of executable content on the Internet, introducing program code that can be transported along with state information. Agents are objects that can move from one host on the Internet to another. That is, an agent that executes on one host can suddenly halt execution, dispatch itself to a remote host, and resume execution there. When the agent moves, it takes along its program code as well as its data. It is an application that can migrate from host to host in a network, at times and to places of their own choosing. The state of the running programme is saved, transported to the new host, and restored, allowing the programme to continue where it left off.

Agents support the concepts of autonomous execution and dynamic routing on their itinerary. You can also think of the agent as a generalization and extension of Java applets and servlets. Agents are hosted by an agent server or agent system in a way similar to the way applets are hosted by a web browser. The agent system provides an environment for agents, and the Java virtual machine

and the security manager makes it safe to receive and host agents. Mobile agents differ from **applets**, which are programmes downloaded as the result of a user action, then executed from beginning to end on the host. With every new access on an applet it starts at the beginning, runs till the end, can't move from one users host to another and can't save its state. If these characteristics aren't given [Kotz99] talks about mobile code. Mobile agents are an effective choice for many applications, including improvements in latency and bandwidth of client-server applications and reducing vulnerability to network disconnection. There are several trends affecting Internet technology and activity. The bandwidth to many end users will remain limited by several technical factors. Many users will still connect via modem over a copper loop. An area with an enormous growth in the computer industry is that of portable computing devices. Web based services and web terminals will become commonplace in public places.

Now let me mention some words about possible **employments** of this technique. Mobile agents seem suitable for applications, such as electronic commerce, system administration, network management, and information retrieval. Another domain is disconnecting computing, such as laptops and personal digital assistants that are frequently disconnected from the network or use a wireless network that might become interrupted on short notice. The agent could be a permanent representation of a user on a remote server.

Independent of the way agents are used, the **goals** are nearly the same. These are the reduction of the network traffic and an asynchronous interaction.

2.1.2 The Agent System

A **mobile agent system** is the platform or the middleware for all agents – a uniform platform that hides the underlying machines. Mobile agent systems differ from process migration systems in that the agents move when they choose, typically through, a 'jump' or 'go' statement, whereas in a **process migration system** the system decides when and where to move the running process (typically to balance CPU load).

The system also consists of:

- an **API** development kit, a set or library of classes and interfaces that allows you to create mobile agents. It is also used to interpret arriving agent code.

- a scripting language system with interpreter. The agents are implemented in a programming language. Because of the hardware independency Java is mostly used.

- runtime support and operating system services accessible via a scripting language.

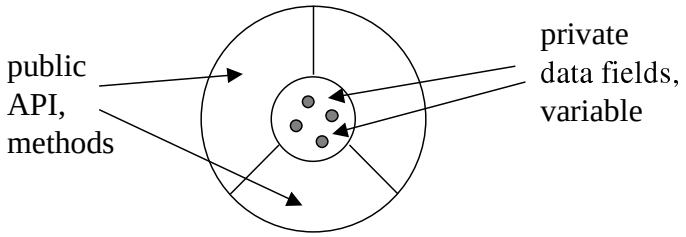
Current mobile agent systems save network latency and bandwidth at the expense of higher loads on the service machines, since agents are often written in a relatively slow interpreted language for portability and security reasons, and since the agents must be injected into an appropriate execution environment upon arrival.

2.1.3 Object-Oriented Approach

What is an object?

You can look around you now and see many examples of real-world objects. These real-world objects share two characteristics: they all have state and they all have behaviour. For example, cars have state (current gear, number of gears, maximum speed) and behaviour (braking, accelerating, changing gear). Another example is an abstract example. An event is a common object used in GUI window systems to represent the action of a user pressing a mouse button.

Software objects are modelled after real-world objects in that they have state and behaviour, too. A software object maintains its state in variables and implements its behaviour with methods.



As you can see from the diagram, the object's variables make up the centre of the object. Methods surround and hide the objects data fields from other objects in the program. Packaging an object's variables within the protective custody of its methods is called encapsulation. The benefits of encapsulation are modularity and

information hiding. Objects interact with one another by sending each other messages.

It is possible to have many objects of the same kind that share characteristics. They are similar and you can create a blueprint for those objects. These patterns or prototype are called **classes** and define the variables and methods. Classes provide the benefit of reusability. You can build a relation ship between classes. With inheritance and aggregation you can transfer functionality from one class to another.

2.1.4 Programming Language Java

It would seem that the general acceptance of Java as the de facto Internet programming language gives rise to accelerated research in mobile agents.

The most popular slogan of Java is ‘write once, run anywhere’! That is, once you have written Java code, for example an agent, it will run on every machine that supports the agent API. You need not be concerned with the underlying hardware or operating system or with the nature of the particular implementation.

Java is an object-oriented programming language. Its advantages are that it is distributable, robust, secure, multi threaded, dynamic. It is the ideal language to implement mobile agents because it is platform independent. The compiler generates Java bytecode running on a virtual machine. The virtual machines are available for a lot of operating systems.

2.2 Initial Survey

2.2.1 Agent Shaping

If we think about our daily life, sometimes we come across the title ‘agent’. An agent is a person that assists, consults, manages, observes, collects, spies, snoops and searches for other people. All these skills are totally matching to software agents. They act on the users behalf, too. Agents perform by allowing people to delegate work to them.

Mobility is not a mandatory property of agents. A **stationary agent** executes on the system where it begins execution and is bound to that system. If it needs information that is not on that system or needs to interact with an agent on a different system it generates a mobile agent. A **mobile agent** has the unique ability to transport itself, its state and code from one system in a network to another where it resumes execution. By the term **state**, I mean the attribute values of the agent. The **code** in an object-oriented context is the class code necessary for the agent to execute.

The **range** of the agent technology is described in [Karm98]. At one end of the scale are relatively simple, client-based software applications that can assist users in performing mundane tasks such as sorting e-mail or downloading web pages. This class of agents are often referred to as 'personal assistant' agents. At the other end of the scale is the concept of sophisticated software entities possessing artificial intelligence that autonomously travel through a network environment and make complex decisions on a user's behalf.

2.2.2 Common Environment

The numerous available agent systems have great differences in their characteristics. But if you look at all applications, you would find a property shared by all agents: the fact that they live in an environment.

The agent system offers the environment for the agents. The first topic I will prepare for my dissertation in the following weeks is a comparison of usable and available agent systems. The result of that will be the choice of the system I will employ.

A mobile agent system is specialized application software that runs on top of an OS to provide MA functionalities. Another forward-looking approach implements agent system requirements as OS extensions to take advantage of existing OS features and the user doesn't have to start the agent system separately.

Aglets

The agent system 'Aglets' models the mobile agents to closely follow the applet model of Java. It is a simple framework where the programmer overrides predefined methods to add desired functionality. An Aglet is defined as a mobile

Java object that visits Aglet-enabled hosts in a computer network. An agent runs in its own thread of execution after arriving at the host, so it is attributed as autonomous. It is also reactive because it responds to incoming messages. The complete ‘Aglets’ object mode includes additional abstractions such as context, proxy, message, itinerary, and identifier. These additional abstractions provide ‘Aglets’ the environment in which it can carry out its tasks. ‘Aglets’ uses a simple proxy object to relay messages and has a message class to encapsulate message exchange between agents. Modelling the MA as a Java object, the designers of ‘Aglets’ leverage the existing Java infrastructure to take care of platform dependent issues and to use existing mobile code facility of Java.

Grasshopper

‘Grasshopper’ builds a region over all hosts in the network with the system on it. Every single host is regarded as an agency with places to hold agents. It is the first environment, which is compliant with the OMG-MASIF standard. The standardisation ensures that your agent applications will be open towards other agent environments and save your investments for the future. The core features of this system are communication, management, persistence, transport, security, and registration. The ‘Grasshopper’ platform code is organised by means of various Java packages that can be subdivided into two main groups. One comprises classes that realise the interface definition language IDL specifications of OMG MASIF and the other comprises ‘Grasshopper’-specific classes to implement the application.

Voyager

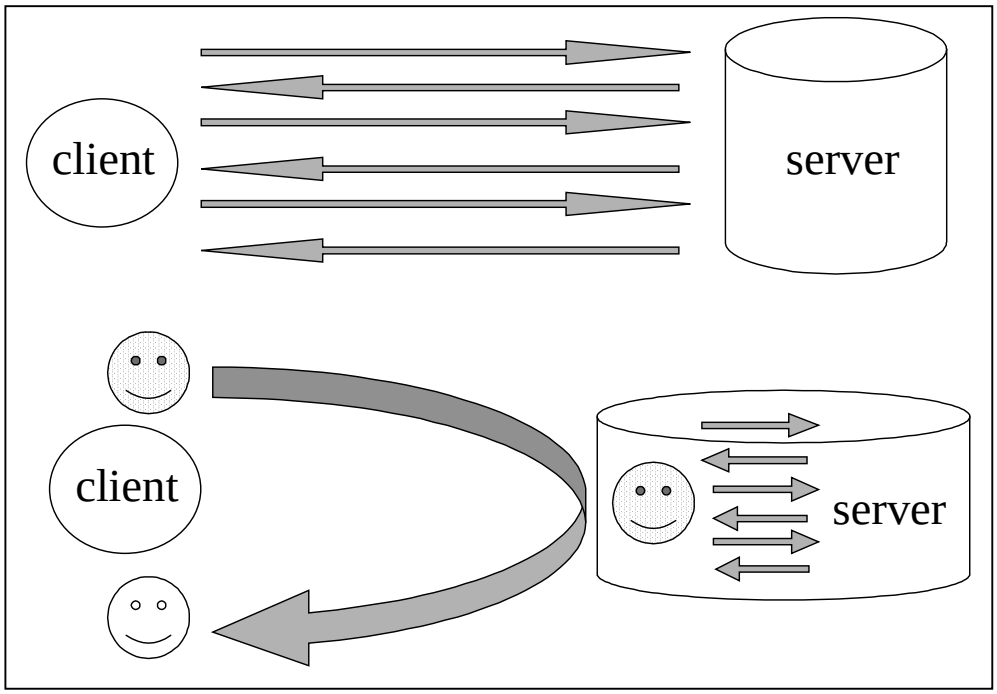
Proclaimed as an agent-enhanced object request broker ORB in Java, ‘Voyager’ offers several advanced mechanisms that could be used to implement MA systems. The voyager agent model is also based on the concept of a collection of Java Objects; it does have an Agent class that developers can subclass to implement ‘Voyager’-style MA. The ‘Voyager’ agent is designed to take advantage of the ‘Voyager’ ORB features, which make extensive use of Java’s reflection mechanism. A ‘Voyager’ agent can communicate by calling methods or using ‘Voyager’ Space technology, a group event and message multicast facility that ObjectSpace claims to be more scalable than simple communication mechanisms.

In summary, MA systems consist of either Java class libraries, scripting language systems with interpreter and runtime support or OS services accessible via scripting language. I will limit to them offering a Java library. My comparison between the agent systems mentioned before has the following categories: security, portability, mobility, communication, and data management.

2.2.3 Client/Server versus Mobile Agents

In that part of the survey I show the differences of the client/server paradigm and the agent technology.

Traditional client/server systems provide services via centralized nodes known as service control points. The location of the service has to be configured on the client to set up the connection. If the service is provided from another host, the configuration would be changed. That isn't necessary in an agent system, because the location of a service is held in a registry and is retrieved with every move.



Normally clients and servers communicate via a remote procedure call or a socket connection. The first illustration shows the requests and response between a client and a server. All requests start at the client and move over the network to the server. The result is sent back to the client. That scenario is necessary for every request. The mobile agent approach of the second illustration starts an agent with all required information. The agent moves to the server and collects the response.

The interaction between the client and the server is shifted from the network to the server host. That means that the network traffic could be reduced and the execution is much faster.

The fundamental difference is the communication. If a mobile agent needs data of another computer, he would initiate a request. He migrates to the foreign host. At that location he will start the execution at that point where he stopped it. Only if all tasks are done, the agent moves back to the home computer.

2.3 Aims and Objectives

The aim of this project is to develop a flight booking system for an independent search for offered flights and hotel rooms.

The overall aims of the proposed project are:

- To provide a user with the possibility of a graphical user interface to book a flight independently of a special travel agency or a special offer.
- To give all travel agencies the opportunity to offer their flights to every one who is interested in.
- To book a hotel room in conjunction with the flight. The travel agencies have special prices for the hotel rooms.
- To compare the search results of the different offers and give the user the possibility to choose one.
- To define security requirements for the machines on which an agent moves and how the data protection on foreign machines is possible.
- To define resource protection against non authorized access to the data of the agent

3 Time-Plan

The following sections describe the thoughts about structuring and organizing the project.

The main part of the dissertation is the design and implementation of the application using mobile agents. This is a software project. That means, that there will be a walk through the phases defined in software engineering.

The master project begins after the exam week and ends at the end of March 2001. In this period of time will be a longer break because of the project in my company. During these 19 weeks the background knowledge will be completed, the interim report will be written, agent systems will be compared, one system will be chosen and the software lifecycle for developing the application will be realized.

The object-oriented paradigm of software engineering consists of seven phases. These phases are:

- requirements
- analysis
- design
- programming
- integration
- maintenance
- retirement

The duration of a phase is calculated with the approximate average percentage of time spent on the development phases given in [Scha99]. The timetable shows the percentage for each phase. The project begins with the requirements phase on the 15th July and ends on the 24th February with the last tests. These are fourteen weeks for the application.

The following table shows the division of the weeks and the structure of the phases. The third column shows the weeks for every phase calculated with the given percentages.

	percentage of time	weeks for one phase
Requirements and Analysis	22 %	3 weeks
Design phase	19 %	3 weeks
Implementation phase	36 %	5 weeks
Integration, test phase	23 %	3 weeks

[Scha99] puts the requirement phase and the analysis phase in one block. There is one week to specify the requirements and 2 weeks to spend on the analysis. Maintenance and retirement are two phases of a professional project and are located after the integration and test phase. Maintenance is very important to support the customer when he uses the software.

3.1 Phases

My timetable begins with the background phase. This period started in April when I collected the books, journals, reports, abstracts and so on about my topic.

Until the 3rd July I finish my interim report with an introduction and an overview of the project topic and a background chapter with an initial survey, with the aims and objectives and with the deliverables.

To use mobile agents in a network, you need an agent system, which provides the application programming interface (API) and the communication infrastructure. There are different agent systems from different vendors. The first task to do is a survey on these systems with the decision which one to use. The six steps of the software development process can begin after this choice.

The project starts with one week to set up the requirements. The task of the developer at this stage is to determine exactly what the client needs and to find what constraints exist. All significant requirements are structured in a top down view.

The object oriented analysis phase consists of three steps. The first is the use case modelling and associated scenarios. The use cases show what the user can do with the software. Scenarios are specific instantiations of the use case. Sufficient scenarios should be studied to have a comprehensive insight into the behaviour of the system being modelled. This information will be used in the

next phase to determine the objects. The classes and their attributes are determined in the second phase - class modelling. The interrelationship between the classes is shown in a class diagram. This phase ends with the dynamic modelling.

The design can start with the results of the previous phase. The topics of this phase are the construction of interaction diagrams for each scenario and the detailed class diagram with the methods for each class. This is followed by the design of the product in terms of objects.

The implementation phase lasts 5 weeks. The tasks to do in that time are to create a stationary agent starting a mobile agent moving through the network, to create a stationary agent to provide the flight information and a second agent handling the requests for the hotel rooms. The last action is the combination of these parts with the web technology. There is a division of this phase into three parts. Two weeks for the implementation of stationary agents, two weeks for mobile agents and another week for the web technology.

The integration of the source code starts in the implementation phase and is tightly coupled with the tests.

The last phase is the test phase during which test cases are set up to test different scenarios to see, that the software application reacts like it should.

A rapid prototype is a piece of software hurriedly put together that incorporates some of the functionality of the target product but omits those aspects generally invisible to the client like storing data in a file or accessing a database. It is principally a graphical user interface with rudimentary functionality. The prototype is useful for experiments and can be changed. It tells all involved persons whether the final application meets the needs and it tells the developer if special requirements are possible or not. The development of the prototype will start in the requirement phase and it is used in the analysis phase to see if some aspects are possible or not.

The documentation will be made successive after every phase and at the end of the project. It is much more useful to write a detailed documentation during every phase. The main documentation periods are the report, the comparison, the requirements, the design, the implementation and the test phase.

4 Deliverables or specific outcomes

The milestones are the deadlines for the deliverables. In the timetable you can find them indicated with M and a number.

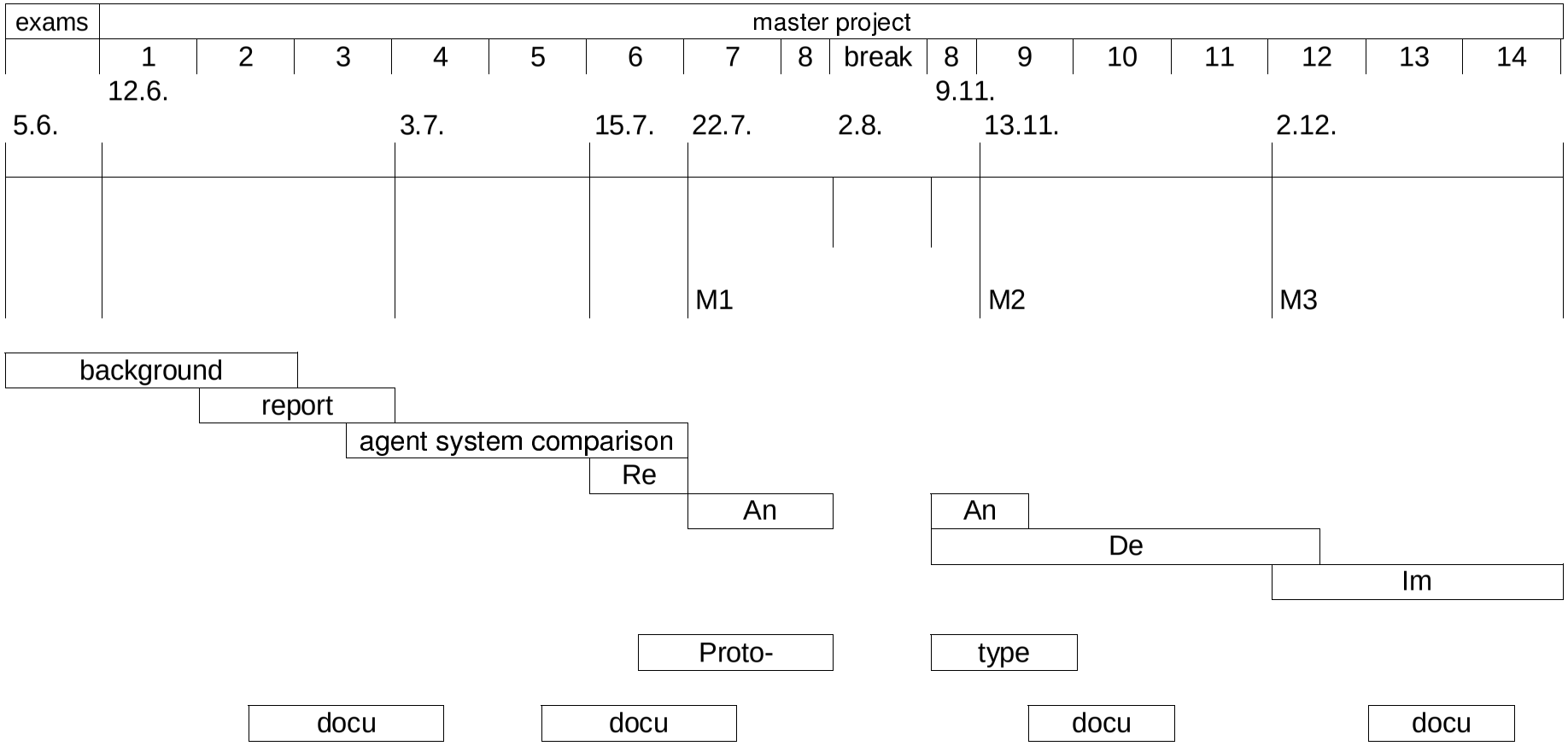
M1 submission of the comparison of agent systems and the requirements

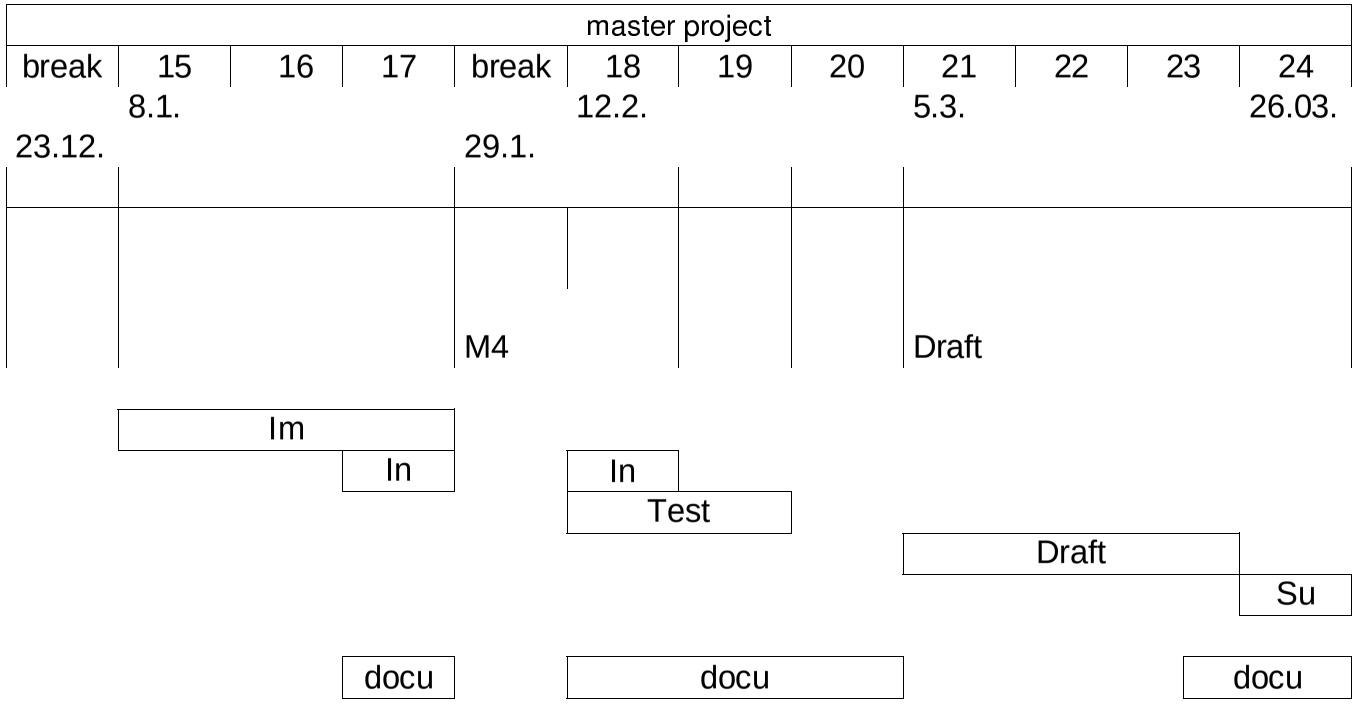
M2 submission of the object oriented analysis phase

M3 submission of the object oriented design phase

M4 submission of the implementation

draft submission of the final draft version


exams
master project
1
2
3
4
5
6
7
8
break
8
9
10
11
12
13
14
12.6.
9.11.
5.6.
3.7.
15.7.
22.7.
2.8.
13.11.
2.12.
M1
M2
M3
background
report
agent system comparison
Re
An
An
De
Im
Proto-
type
docu
docu
docu
docu


master project
break
15
16
17
break
18
19
20
21
22
23
24
8.1.
12.2.
5.3.
26.03.
23.12.
29.1.
M4
Draft
Im
In
In
Test
Draft
Su
docu
docu
docu