Fruitlets - a Kind of Mobile Component
Master's Thesis

Juerg Gertsch

January 27, 1997

Abstract

Keywords: Mobile Code, World-Wide-Web, Software Composition, Java.

Mobile software entities are becoming increasingly important in the domain of local area networks (LAN) and wide area networks (WAN). Different kinds of mobile entities are a rapidly evolving area of research in the field of World-Wide-Web, distributed and open systems. The first part of this thesis surveys different approaches in order to develop open, flexible and distributed systems. We focus on an approach of stateless mobile software entities. The second part of this thesis introduces the notion "fruitlet" and "run-time framework" and classifies "fruitlet" within other existing mobility meanings and compares them with other concepts related to mobile code. We describe a prototype of a basic software architecture for stateless mobile software entities. The third part shows examples of open, flexible and extendable software applications using the described technology of "fruitlets".

Contents

1 Introduction **7**

1.1 Introduction 7

1.2 Goal 8

1.3 Overview 8

2 Problem Analysis **9**

2.1 Current Situation 9

2.1.1 New Application Domains 9

2.1.2 Technological Innovations 10

2.2 Problems 11

2.3 Requirements 11

2.3.1 Mobility and Heterogeneity 11

2.3.2 Adaptability 12

2.3.3 Security 12

2.4 Conclusion 13

3 Available Technologies **14**

3.1 Object-Oriented Technologies 14

3.2 Microsoft's Component Object Model 15

3.2.1 COM Components 16

3.2.2 COM Interoperability 16

3.2.3 COM Local/Remote Transparency 16

3.3 CORBA 16

3.3.1 CORBA Component 17

3.3.2 CORBA Interoperability 17
3.3.3 Conclusion COM/CORBA 18
3.4 Applets and Java Enabled Web Browsers 19
3.4.1 Applets 19
3.4.2 Application Domains of Applets 20
3.4.3 Applets and Security 21
3.5 Mobility 22
3.5.1 Mobile Code Representation 22
3.5.2 Mobile Stateful Software Entities 23
3.5.3 Mobile Stateless Software Entities 23
3.5.4 Security and Mobile Code 24
3.6 Conclusion: Where to Go From Here 25

4 Programming Languages for Mobile Programming, a Comparison 26
4.1 Obliq 27
4.1.1 Visual Obliq 28
4.1.2 Discussion 28
4.2 Telescript 28
4.2.1 Language Characteristics 29
4.2.2 Language Concepts 29
4.2.3 Class Libraries 30
4.2.4 Discussion 30
4.3 Safe-Tcl/Incr Tcl 30
4.3.1 Discussion 31
4.4 Java 31
4.4.1 Object-Oriented Properties 32
4.4.2 Security 32
4.4.3 Class Libraries 32
4.4.4 Discussion 32
4.5 Conclusion 32

5 Model **34**

5.1 Definitions . . . 34

5.1.1 Component . . . 34

5.1.2 Mobile Component . . . 35

5.2 Proposed Model . . . 35

5.2.1 Fruitlet . . . 35

5.2.2 Plugs of Fruitlets . . . 36

5.2.3 Naming and Locating Fruitlets . . . 37

5.2.4 Run-Time Framework . . . 37

5.2.5 Security Impacts . . . 37

5.3 Conclusion . . . 38

6 Prototype **39**

6.1 The Prototype meaning of "Fruitlet" and "Run-Time Framework" . . . 39

6.2 Design . . . 40

6.2.1 ManagerIfc . . . 40

6.2.2 ComponentIfc . . . 41

6.2.3 Fruitlet Parameterization . . . 43

6.2.4 Security . . . 43

6.2.5 Component Loader . . . 46

6.3 Prototype Packages . . . 46

6.3.1 Class Diagram of Package `unibe.componentloader` . . . 47

6.3.2 Class Diagram of Package `unibe.net.http` . . . 47

6.3.3 Package `unibe.security` . . . 48

6.3.4 Class Diagram of Package `unibe.run` . . . 48

7 Use Cases **49**

7.1 General Internet Server . . . 49

7.1.1 Design . . . 50

7.2 HTTP Server . . . 50

7.2.1 Design . . . 50

7.2.2 Conclusion: HTTP Server 51
7.3 Open Message Organizer 51
7.3.1 Design 52
7.3.2 Omo Run-time Connector/Composer 52
7.3.3 The Ada 95 Experiment 53
7.3.4 Conclusion: Open Message Organizer 54
7.4 Conclusion 54

8 Conclusion 55

8.1 Conclusion 55
8.1.1 Adaptability 55
8.1.2 Heterogeneity 56
8.1.3 Mobility 56
8.1.4 Security 56
8.1.5 Fruitlet Granularity 56
8.2 Open Problems 57
8.2.1 Security Concept 57
8.2.2 Performance 57
8.3 Further Work 57
8.3.1 Run-time Composition 57
8.3.2 Synchronization during Run-time 58
8.3.3 New Trend: Java Beans 58
8.4 Acknowledgments 58

A Use Cases: Design 59

A.1 General Internet Server 59
A.1.1 Design 59
A.1.2 ServerIfc 60
A.1.3 ServiceRequestIfc 61
A.2 HTTP Server 62
A.2.1 Design 62

A.2.2 HTTPTaskIfc 64
A.2.3 HTTPRequestIfc 64
A.2.4 HTTPResolveIfc 65
A.2.5 HTTPResponseIfc 66
A.2.6 Conclusion: HTTP Server 67
A.3 Open Message Organizer 68
A.3.1 Design 68
A.3.2 Interfaces 69
A.3.3 Examples of Omo fruitlets 81
A.3.4 Conclusion: Open Message Organizer 84

B A Closer Look at Java 85

B.1 Introduction 85
B.2 Basic Concepts of Java 86
B.3 The Java Programming Language 86
B.3.1 Java's Object Model 86
B.3.2 Encapsulation 87
B.3.3 Inheritance 87
B.3.4 Polymorphism 87
B.3.5 Dynamism 87
B.4 The Java Virtual Machine 88
B.5 Concurrency 88
B.6 Class Loading Mechanism 88
B.7 Safety and Security 89
B.7.1 Language and Compiler 89
B.7.2 The Byte-Code Verifier 89
B.7.3 The Class Loader 90
B.7.4 Security Manager 90
B.8 The Java Libraries 91
B.8.1 java.lang 91
B.8.2 java.net 91

B.8.3 java.awt 91
B.8.4 java.io 91
B.8.5 java.util 92
B.8.6 java.applet 92

C Diagram Notation **93**

C.1 Introduction 93
C.2 Classes 93
C.3 Association 93
C.4 Inheritance 94
C.5 Interfaces 95
C.5.1 Reified Interface Notation 95
C.5.2 Symbolic Interface Notation 95

Chapter 1

Introduction

1.1 Introduction

Current software systems have to satisfy great demands in various application domains. We need applications which can be easily extended, which are easy to adapt to new requirements, and which can run in widely distributed networking environments, like the Internet. In addition, applications should be platform-independent, reusable, easy to develop and maintain.

The tremendous growth of the Internet, especially the popularity of the World-Wide-Web service led to new situations and problems of software applications. Thus, we can observe a new application domain of software entities which appeared in the last few years and which poses for new requirements.

This thesis deals with an approach in the field of mobile code systems within networking environments. We discuss special stateless mobile software entities and describe a basic software layer in order to deal with such stateless mobile components.

The discussion about general problems and requirements leads us to a survey of available technologies in the field. Afterward, we restrict ourselves to a kind of stateless mobile components and we define properties and demands of such components.

The remaining part of the thesis deals with the implementation of a technology for our stateless mobile components. We use the Java programming language [Sun95a] as the implementation language. We show some examples of applications developed using our approach. Such applications are dynamically extendable in a restricted way during run-time and startup-time with stateless mobile components. Furthermore, such applications can also exchange software parts under some circumstances during run-time.

1.2 Goal

We would like to analyze some problems within the current situation of distributed networking systems focusing on the World-Wide-Web. Furthermore, we give a review of some related technology concerning our requirements.

The prototype implementation of an open run-time architecture and the explanation of stateless mobile components shows a development approach of open, flexible applications, especially within the development area of open servers and open browsers.

1.3 Overview

In chapter 2 we discuss some problems of current software applications, mainly in the field of World-Wide-Web. Next, chapter 3 shows and discusses available technologies in our field of interest and related ideas and projects. Chapter 4 gives a comparison between four programming languages in the field of mobile code systems. In chapter 5 we give definitions and introduce the notions "fruitlet" and "run-time framework". We discuss a set of properties of these notions. Chapter 6 is about a prototype "run-time framework" which we implement in Java. In chapter 7 we show examples running within our prototype "run-time framework" and we present our conclusions in chapter 8.

Chapter 2

Problem Analysis

This section focuses on some view points of the current situation of software evolution related to the World-Wide-Web (Web for short) field. The first part of the chapter describes the current situation of the Web application domain. The second part gives an overview of problems which arise in this field and the last part lists some requirements of applications based on distributed networking environments like the Web.

2.1 Current Situation

2.1.1 New Application Domains

The tremendous growth of wide area networks (WAN for short) in the last few years, led to different new application domains of software applications. Especially the Internet gained an incredible importance around the globe. Within the Internet different services like electronic mail, news groups, file transfer, telnet, and the World-Wide-Web became very popular. Since three or four years the World-Wide-Web service is undoubtably the most popular service on the Internet. The Web was originally developed as a distributed hypertext information system on top of the Internet. Because of the easy-to-use handling it caused an explosion of new available hypertext information on the Internet and thus an explosion of the Internet itself. Information about every imaginable (and also unimaginable) topic appeared over the globe.

Moreover, the Web also became the role of an integration technology of other Internet services. The Web is now able to transfer files and to perform database queries as well as to send electronic mail messages around the globe.

In short, the Web introduced a new platform of distributed computing (we will also refer to this new platform as "new application domain") on top of the Internet. This platform shows the following characteristics:

- Clients participating within this new application domain are working on many different platforms. Thus, software tools (e.g. browsers, servers) must be able to run in

a distributed *heterogeneous* networking environment. This networking environment consists of arbitrary platforms running different windowing systems like Macintosh, MS Windows and UNIX derivates with X windowing systems.

- The community taking part in this application domain is firstly *very large* and secondly *distributed* over the world.

- The technology in the Web application domain is quite *unstable* and technological environments *rapidly evolves* in new directions. The standards in this application domain evolve very fast and became outdated in a very short period of time. In addition, these new standards and technologies are rapidly introduced and distributed to the end user community.

- Most of the users in the application domain of the Web are *non computer professionals* (e.g. journalists, marketing specialists, advertising specialists, web publishers etc.). Thus, technologies and standards have to take care about the *different knowledge* of users within the Web application domain.

2.1.2 Technological Innovations

There are a lot of technological innovations in the field of the World-Wide-Web concerning protocols, browser techniques, server design, security techniques and techniques related to integration efforts. One interesting and innovative field in the application domain of the Web and the networking application domain in general, is the field of mobile code systems. Along the wealth of innovations we want to focus here on mobile code systems.

Various systems and technologies related to mobile code systems appeared in the last few years. Most of them are based on a transportable, platform independent code representation, which allows for execution on the client side within a safe compartment. Popular examples of mobile code systems are Java[1], Penguin[2] and Safe-Tcl [BR93].

Java applets for instance, enables Web browsers to render active documents. Parts of such active documents may be controlled by imported mobile Java code (see section 3.4). In the Web application domain, Java applets overcome the limited field of passive documents and open the doors to active documents. Thus, Java applets introduce technologies like animation and sophisticated user interfaces to the Web application domain (see also section 3.4.2). In contrast to common software applications, applets show examples of mini "instant" applications (or parts of applications) which are able to run in a Web browser environment without explicit installation steps (the installation process is reduced to the "binding" process of an applet into the well defined browser environment). Furthermore, applets fit well into the simple browsing scheme of current Web browsers.

[1] Java: `http://java.sun.com/`

[2] Penguin: `http://www.eden.com/~fsg/penguin.html`

2.2 Problems

As described in the previous section, the current situation leads to new application domains. Within these new application domains we can recognize new (and old) problems. Besides technical problems like network capacity problems, transport medium problems, protocol problems the Internet also causes quite a few social problems. In this thesis we focus on technological problems, more precise on problems related to software technology.

We feel that the following list of problems gives an interesting overview.

- Applications (and parts of applications) become quickly outdated because of the technological evolution in the field. Rapidly changing standards and protocols lead to obsoleted software components. In a widely distributed environment, outdated applications cause new integration problems and interoperability problems.
- The missing portability of applications leads to an obstacle of the participants in the networking environment. Thus, standard applications and services must be ported to every popular platform.
- Applications or parts of applications normally need a big installation effort on a particular system.
- Applications (and parts of applications) cannot be moved within a distributed environment. Applications lack suitable code representations and mechanisms to transport them.
- The procedure of down-loading and installing applications from the Internet is commonly in use. The problem remains that such code is not always trustworthy. We never exactly know if the code affects our system in a dangerous way.

2.3 Requirements

In consideration of the discussion of current problems in the previous section, we try to declare requirements of current applications in the field of the World-Wide-Web and the Internet. Although these requirements cannot lead to the perfect solution, we hope to gain some improvements in the mentioned field. The previous list shows that *portability* and *movability* are quite related (we use *movability* and *mobility* interchangeably). Portability can even be regarded as a kind of mobility between different platforms (see section 3.5). Thus, we see mobility as a central requirement of applications within the field of widely distributed networking environments. Mobile components can help to improve the current status of the Web application domain.

2.3.1 Mobility and Heterogeneity

One central requirement we want to focus on is mobility. As a matter of fact a distributed networking environment (e.g. the World-Wide-Web) is the natural platform for mobility.

Mobility can have a lot of meanings and different views in a networking environment (see discussion in section 3.5). We focus on the requirement that network applications should be able to import mobile software entities into their run-time system. This view of mobility leads us also directly to requirements like "platform independency" and "security".

The openness we require from our applications naturally leads to the requirement of platform independent software. We refer to this point as *heterogeneity*. Models like COM and CORBA [Mic95, Obj92, Vin93] (see section 3.2 and 3.3) already hide differences between different platforms and different programming languages. Especially in networking environments, we are forced to design and implement parts and services in a portable way.

2.3.2 Adaptability

We should be able to adapt existing applications in order to fulfill changing requirements of standards and protocols in the field of World-Wide-Web. This naturally leads us to open, flexible applications. It is also imaginable that we need adaptability during run-time. We refer to this point as *reconfigurability* of applications. In the field of open servers and open browsers (especially Web browsers) it is necessary to achieve reconfigurable software applications.

Changing requirements of software applications can also be referred to as *evolution* [ND95].

Adaptability may also be viewed as a form of reusability, because existing structures (e.g. applications) can be reused to create slightly different applications [MN96].

2.3.3 Security

Security has many different facets within computer science and computer security itself is a very large research topic. Important areas of security are for instance *data loss* (hardware errors, software errors) and *protection of system resources*. Because we are interested in mobile code systems, we restrict security to protection of system resources like file system and network access. Throughout the thesis we also use the term security with the meaning of resource protection.

Security is an important problem in the field of mobile code systems, mainly because untrusted code can damage the integrity of a host computer in many ways. Thus, the requirement of *mobility* undoubtably leads to the requirement of *security*. Taking into account that mobile code is often loaded implicitly with documents (e.g. HTML documents), it becomes even more important to protect host computers from untrusted mobile code.

All pieces of software running in a networking environment have to conform to security rules. This requirement will become more and more important in the near future.

2.4 Conclusion

This chapter discussed the current situation and some problems of applications within a distributed networking environment like the World-Wide-Web. We gave a short list of requirements we demand of current applications in distributed networking environments.

In chapter 3 we survey different approaches in the field of open systems, distributed programming and in the World-Wide-Web field, which helps to fulfill particular requirements of the last chapter. Chapter 3 gives also an overview of the central notion "mobility".

In fact, section 3.6 concludes chapter 2 and chapter 3. At this point we shall give an outline of the remaining chapters of this thesis.

Chapter 3

Available Technologies

This chapter contains an overview of solutions related to the problems discussed in chapter 2. We choose mainly technologies which are relevant in our domain of interest. This means networking environments, mobile code, World-Wide-Web and software composition.

The second part of the chapter contains a discussion about the notion "mobility". We discuss the various usage of the notion and we finally try to integrate our meaning of mobility into the existing hierarchy of mobility.

3.1 Object-Oriented Technologies

Object-oriented programming is one of the current approaches to solve problems in the software industry. The object model helps us to realize principles like encapsulation, abstraction, modularity and hierarchical decomposition. The object-oriented paradigm can be described as the nineteen eighties successor of structured design and structured programming of the seventies. Object-oriented technology has several advantages. Firstly, the object model leads to reuse possibilities of implementation and design. This property leads to reusable application frameworks. Frameworks provide not only reusable code, but more importantly, they carry a lot of information about design [GHJV95]. Secondly, objects reflect a natural concept in a human point of view. However, object-oriented frameworks often become large, complex class structures. This results in hard to learn and hard to understand class hierarchies and often makes it impossible to specialize a framework by less expierenced framework users. The specialization of an object-oriented class hierarchy is often realized using class inheritance. Inheritance implies a direct implementation dependency between super- and subclasses in an "is-a" relationship manner. If super- and subclasses are maintained by different groups of developers, this fact can lead to serious integration problems. An application framework can cover a well defined problem domain. The combination of two or more different frameworks seems to be very hard, because the different designs often disturb each other. There are some discussions about framework integration in the literature [Hoe93].

Often the frameworks themselves are platform dependent, because most of them are writ-

ten in languages like C++ and deal directly with the operating system and/or graphical user interface on a special platform. Thus, the derived applications are platform dependent anyway.

Object-oriented programming can of course improve the reusability of software and design. Especially, if the software has to run on a specific system and is maintained by a controlled group of developers.

3.2 Microsoft's Component Object Model

The component object model (COM for short) was introduced by Microsoft Corporation in 1995 and provides a solution in the direction of software components [Mic95]. COM[1] was introduced to cover some problems in the software industry and to improve software development. Microsoft gives an overview of the current problems with the software developing and maintaining process:

- Applications are large and complex.
- Applications are monolithic.
- Applications are not easily integrated.
- Programming models are inconsistent.

We can observe some corresponding items to our chapter 2. On the other hand Microsoft lists some requirements focusing on

- Client/server computing
- Object-oriented concepts
- Distributed computing

COM is a basic standard to reuse binary components. The basic standard relies on a client/server model. The advantage is that the usage of all components is transparent in respect to the location. The programmer can refer to any component exactly in the same manner, independently of the location.

COM is not a specification for how applications are structured: it is a specification for how applications interoperate.

[1] COM: `http://www.microsoft.com/intdev/sdk/docs/com/comintro.htm`

3.2.1 COM Components

Microsoft describes the solution as a system, where application developers create reusable software components. They define a component as "a reusable piece of software in *binary* form that can be plugged into other components from other vendors with relatively little effort." COM is an object-based programming model designed to promote software interoperability.

COM provides a binary standard and a network standard to enable interoperability between applications (or parts of applications) developed by different vendors or companies. Applications interact through a defined set of functions. We call a set of function signatures an *interface*. An interface in this sense is a strongly typed contract between software components. An interface is an articulation of expected behavior and responsibilities of software components.

3.2.2 COM Interoperability

In COM, the interaction between objects and the users of those objects is based on the client/server paradigm. All objects, which a client can access live on a server site. This server can be a *In-Process Server* a *Local Server* or a *Remote Server*. A client can ask a server to create a new object and the server returns an interface to the created object.

3.2.3 COM Local/Remote Transparency

The binary standard allows COM to intercept a method call and to handle the call in a transparent fashion. COM makes an RPC (Remote Procedure Call) to the real instance of an object, which can be running in another process or even on another machine. From the developers point of view there is no difference between local and remote objects. Thus, the developer can use the COM objects in a transparent fashion with respect to the location of a COM object.

One problem of local/remote transparency is to determine the life-cycle of objects. Clients of COM objects do not know about the location of a concrete object. The server of an object has to deal with reference counting in order to notice whether an object can be deleted.

COM objects are not mobile entities. COM itself is purely a remote object technology, but there are some technologies on top of COM which allows for mobile code systems (e.g. ActiveX components[2]).

3.3 CORBA

CORBA stands for Common Object Request Broker Architecture and is a specification from OMG [Obj92, Vin93]. CORBA specifies a standard which provides interoperabil-

[2]ActiveX: `http://www.microsoft.com/ActiveX/`

ity between objects in a heterogeneous, distributed, object-oriented environment. Thus, CORBA allows the access of distributed objects in a networking environment. Similar to the COM specification (see section 3.2), CORBA provides services of remote objects through interfaces (in contrast to COM, CORBA objects can only implement one interface). CORBA is also called an *Object Bus*.

At the time of writing, CORBA does not have any specifications about security.

3.3.1 CORBA Component

A client program can invoke an operation of an object by requesting the operation through the ORB (Object Request Broker). The ORB has to manage the transfer of request messages and response messages between the involved objects. The ORB must define a standard network representation in order to transmit the CORBA data types. Objects themselves are never transmitted, just object references.

The interface definition language (IDL) is used to describe and define object interfaces in a way that is independent of the programming language used to implement the objects. IDL defines a set of built-in data types like short, unsigned short, unsigned long, char, unsigned char, octet, boolean and string. IDL allows interface names to be used as types. New interfaces can be derived from other interfaces, multiple inheritance is possible.

A tool called the IDL compiler generates code in order to use at the client side and the server side respectively. These fragments of code are called *stubs* for the client and *skeletons* for the server.

3.3.2 CORBA Interoperability

Like in the Component Object Model (see section 3.2), the interaction between objects is always based on the client/server paradigm. All objects, which a CORBA client can access live on a server site.

CORBA maintains an interface repository and an implementation repository. The interface repository collects run-time information on the IDL interfaces, while the implementation repository is a run-time data structure that can be queried to discover what classes a server supports or what objects are instantiated. CORBA can also assemble a method invocation dynamically, during run-time, via the Dynamic Invocation Interface (DII).

CORBA can be referred to as an interoperability standard. The standard overcomes programming language boundaries and hides the exact locations of objects. This leads to a homogeneous, transparent programming model concerning the object instances.

Figure 3.1 shows an overview of the basic CORBA architecture.

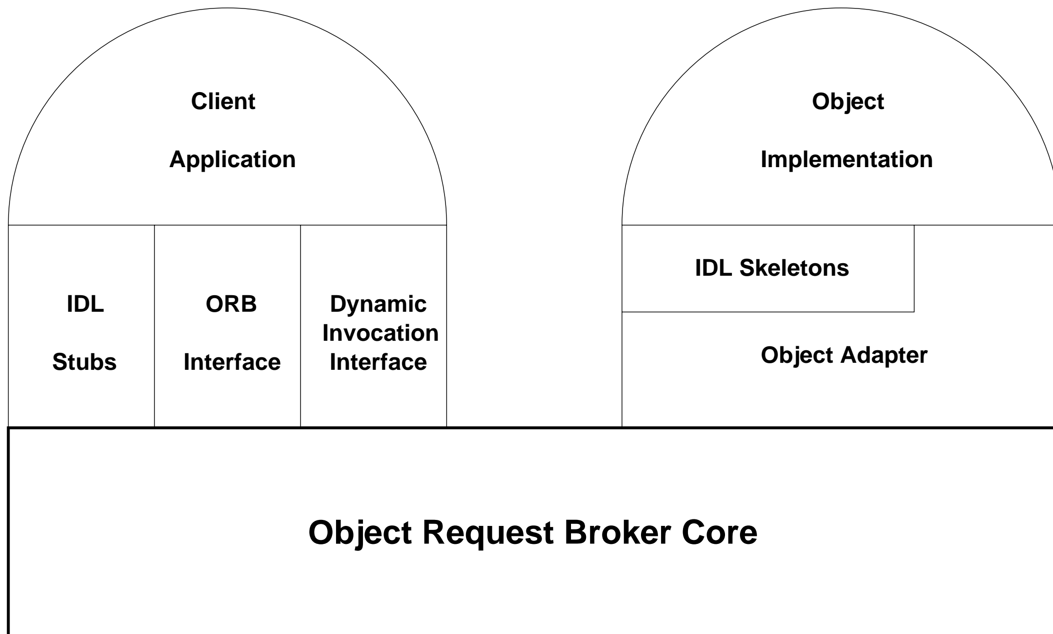

Figure 3.1: Basic CORBA Architecture

3.3.3 Conclusion COM/CORBA

The approaches of CORBA and COM realize the idea of platform and programming language independency of software parts by breaking down the binding of method calls (usually platform and language dependent method calls). An underlying system uses communication to call methods of distributed objects (Remote Procedure Call). The used mechanisms in both cases are quite similar. Objects interact through interfaces, where an interface represents a strongly typed contract between software components. Both approaches are very useful to reuse software (or software parts), because a robustly developed component can later be accessed by new pieces of software using the same technology.

COM and CORBA do not really address mobility in the sense we discuss in section 3.5. Both approaches define interoperability standards and they do not define standards with respect to code representation or mobility.

Further technologies in the field of distributed object models with similar characteristics as COM/CORBA are for instance SOM, DSOM, DOE, and DCE.

A comparison between Java, CORBA, and DCE (Distributed Computing Environment) can be found in [WJK96a].

3.4 Applets and Java Enabled Web Browsers

This section deals with a current example of mobile code in the World-Wide-Web domain. The Web is at the moment a playground for new technologies, especially in the domain of active documents, mobile code and software development in general. As Sun Microsystems introduced Java in the early 1995, the magic word "applet" appeared in the Word-Wide-Web domain. HotJava is the name of the first Java enabled browser, also developed by Sun Microsystems. HotJava is fully developed using the Java technology and Sun was able to introduce the applet technology with HotJava. In the World-Wide-Web community a browser is called "Java enabled", if it is able to import, run and render Java applets. At the moment there are three well known browsers, which are Java enabled. Firstly, the already mentioned HotJava browser from Sun Microsystems, secondly the Netscape browser (version 2.0 and later) and thirdly the Microsoft Explorer 3.0 browser. Whilst the HotJava browser is fully implemented in Java technology, the browser from Netscape is written in conventional C/C++ with additional support for Java.

3.4.1 Applets

Applets show a trend to change the client/server model. Instead of accessing a special service on a remote server side, applications tend to load code needed for a special service and run the service directly on the client side.

Applets are small applications written in the programming language Java. Like common documents on the Web, applets are referenced by an URL (Uniform Resource Locator). Exactly like documents, applets are transferred from a Web-Server side to the Web browser client side. In contrast to documents applets are an active part of a particular web page. An applet consists of executable program code. A "Java enabled browser" must be able to run the code of an applet and to render the graphical representation of the applet inside the browser.

The HotJava browser is able to import such mini applications into its run-time system and let them run within its own run-time system. In fact, applets consist of one or more Java classes. The Java technology (see appendix B) enables the browser to extend the run-time type system of the browser with those classes. The browser expects that an applet fits into the class hierarchy and that it has knowledge of several interfaces. Thus, it is possible for a browser to integrate and run applets in a meaningful way. It is especially important that applets fit into the class hierarchy of the graphical user interface, which allows the browser to pass partial control to the applet. The browser is also able to forward events (mouse click, key strokes) to a given applet. The current specification of applets declares a protocol for using such components. This protocol includes mechanisms to initialize, start and stop applets.

During run-time an applet is composed of run-time entities provided by the browser. Therefore, an applet can control a limited subset of the browsers functionality. Figure 3.2 shows as an overview the class diagram of an applet and a browser environment (for an explanation of the diagram notation see appendix C).

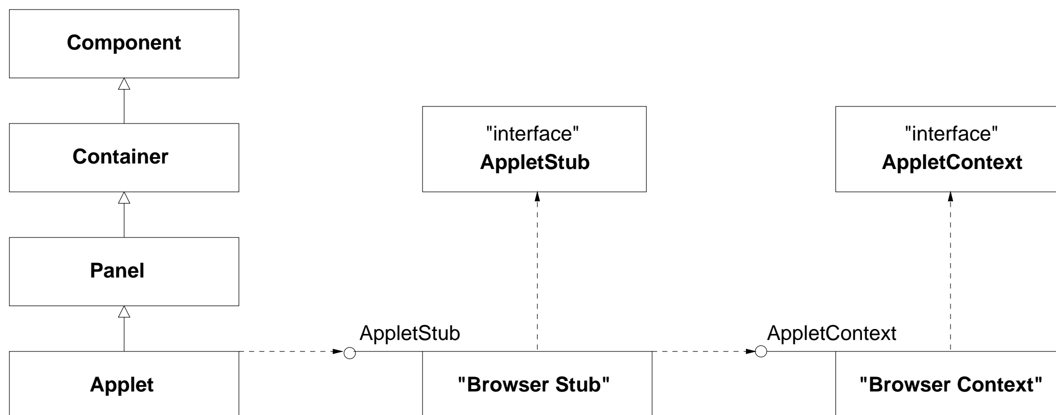

Figure 3.2: Class Diagram: An Applet and its environment

At the moment, applets are web-document based. This means applets belong to a well defined web-page and are rendered (and scrolled) within this page from the browser. If one leaves the page the browser stops the applet.

Applets are special kinds of mobile components (see section 3.5). The browser application can import a description of small components and can integrate such code fragments into the application itself.

The applet approach is of course very limited to the field of World-Wide-Web browsers. The abstraction of an applet is not general enough to use as a general component abstraction to extend and change applications at run-time (see chapter 5).

3.4.2 Application Domains of Applets

In the following sections we show three examples of domains where applets are used.

Multi-Media Applets

Most of the common Web browsers know how to handle embedded graphics in GIF and JPEG format, but they fall short in handling audio and video data directly. Furthermore, they fall short in presenting and controlling multiple streams of multi-media data concurrently.

In the multi-threaded applet environment, it is possible to start and control several threads of multi-media data concurrently. Since applets can react to mouse and keyboard events the user can influence the presentation of multi-media data if the applet provides the respective user interface.

Active Forms

The current Web technology offers passive forms which only provide an interface for entering data. There is a small set of possible elements for forms which can neither be extended nor influenced by the user. This set contains elements for entering text, radio buttons, selectable lists and two types of buttons bound to the actions *send* and *reset form*. The possible elements are part of HTML. No actions can be performed unless the data entered on the form is sent to the corresponding http server.

Applets give rise to more flexible forms concerning firstly the visual representation and secondly (and more important) the control of events and action flow, which can be handled in a very general way. Thus, applets open the door for active forms and spreadsheet-like active documents in the Web field.

Interfaces to Remote Services

The Web can be interpreted as a globally distributed information system. Information providers (servers) and information consumers (clients) normally interact using common Internet protocols, which are strongly related to different kind of servers.

Applets offer a way to implement graphical user interfaces to remote services in a more flexible way. Since applets are platform-independent and mobile, they can provide remote access from clients on different platforms and different locations. Furthermore, the code which implements an interface does not have to be installed on the local machine, rather it is fetched from the information provider's site when required. Finally, since code never has to be installed permanently, the information provider keeps full control over the code provided to the clients: it is possible to distribute upgrades, bug fixes, and the like, immediately to all clients whenever they request the interface.

3.4.3 Applets and Security

A browser which directly imports mobile code from a network must take care about various security issues (see section 3.5.4). Browsers run imported code in a black-box manner, thus they have no idea what (and how) a specific code fragment will execute. Java-enabled browsers should protect themselves and the computer system from any attack by imported code [WJK96b].

Several design decisions help to lead Java in the direction of a safe language right from the beginning. The Java technology provides several layers of security (see appendix B). Firstly, the language itself offers basic mechanisms to write safe code. Next, on the Java byte-code level, we are able to perform some security checks. Finally, the Java core libraries offer security mechanisms (SecurityManager) to control system calls, file access and network access (see appendix B). Java can be referred to as enabling technology (safe environment) for secure mobile code. There are a lot of criticisms and rumors about Java and security at the moment. Indeed, there are some non-trivial security problems with Java (for more details see [DFW96]). For a general discussion about security and mobile

code refer to section 3.5.4.

Java does not implement any security policy. Every application (e.g. a Java-enabled Web browser) has to implement its own security policies for imported code. Thus, a browser has to implement a security policy for imported applets. This means it has to define the operation range of applets. This could range from restricting local file system access to restricting network access to various hosts.

3.5 Mobility

If we consider movable software entities in networking environments, we can observe mainly two different kinds of such entities. Firstly, there are entities which are able to move the state of their run-time representation to another location. Secondly, there are entities which move templates along the network in order to instantiate new run-time representations on another location and thus they cannot move the state of a run-time structure to different location. We refer to the first kind as *stateful* software entities and to the second kind as *stateless* software entities.

We discuss in this section the meaning of mobility in the networking field. The notion of mobility has a wide range of meaning and the survey in this section can never be exhaustive. We restrict ourselves to the field of software abstractions with code mobility. Of course, a similar survey could be done focusing on mobility of data abstraction.

Firstly, we discuss the meaning related with the term code and code representation. Secondly, we classify mobility in conjunction with software entities in general.

Usually, there are two different views of software mobility. Firstly, we understand mobility with respect to movability between different platforms. And secondly, we understand mobility with respect to movability between different locations in a networking environment. We refer to these as "platform mobility" and "location mobility" respectively.

3.5.1 Mobile Code Representation

In order to construct meaningful mobile abstractions we use a representation of the code which is transportable within a network. In the case of code representation, we can say that "platform mobility" always implies "location mobility". If we only consider location mobility between similar platforms we can use hardware or platform dependent code representations. For instance ActiveX components use this approach by transporting hardware dependent code along the network. With respect to platform mobility we can distinguish between low-level representations and high-level representations.

Low-Level Code Representation

By low-level representation we mean code representations, which are not in a human readable form. For example, the Java byte-code [Sun95b, Jol96] is a real low-level repre-

sentation, which can be executed on a virtual machine. For example, the Telescript engine [Gen95] uses a low-level scripting representation. The Juice[3] technology for instance, uses a optimized tree-shaped program code representation in order to transport programs written in the Oberon[4] programming language. The Juice tree representation can be compiled into native platform code by using fast "just in time" compilers. Also this representation technology can be referred to as low-level.

High-Level Code Representation

We refer to high-level code representation as human readable representations. Thus, this category consists mainly of scripting code representations. Examples are JavaScript[5], Perl[6], Tcl[7], SafeTcl [BR93] and AgentTcl[8].

3.5.2 Mobile Stateful Software Entities

This section discusses software entities, which are movable with their state. The state of a software entity is given by all run-time instantiations, which belong to a specific software entity. This category includes the big field of mobile agents [HCK95, Har95, Hoh95] and movable (stateful) application systems [KL96]. Within this category we can distinguish between "autonomous" and "non-autonomous" mobile stateful software entities. "Autonomous" means that a stateful entity can itself decide to migrate to another location. In the case of a "non-autonomous" stateful entity, the entity itself cannot decide to migrate. An example of "non-autonomous" stateful entities in the domain of electronic mail is "active mail" [Bor92, BR93].

Code representation of mobile stateful entities can be low-level or high-level. For instance, an example of a mobile agent system with low-level code representation is [Hoh95] (see also the MOLE project[9]) and "Aglets" (mobile agents in Java[10]). Examples with high-level code representations are active mail with Safe-Tcl [BR93] and Penguin[11]. One example of a stateful mobile application system written in Obliq is described in [KL96].

3.5.3 Mobile Stateless Software Entities

This section discuss software entities which are not movable with state. Mobile stateless entities are used to generate new instantiations at different locations and/or platforms. The instantiations have a well defined state after the instantiation step. Usually, these instantiations are non-movable.

[3] Juice: http://www.ics.uci.edu/~juice/
[4] Oberon: http://www.ics.uci.edu/~oberon/
[5] JavaScript: http://home.netscape.com/eng/mozilla/Gold/handbook/javascript/index.html
[6] Perl: http://www.perl.com/
[7] Tcl: http://ourworld.compuserve.com/homegape/efjohnson/tcl.htm
[8] AgentTcl: http://www.cs.dartmouth.edu/%7Eagent/
[9] MOLE: http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole.html
[10] Aglets: http://www.ibm.co.jp/trl/project/aglets/
[11] Penguin: http://www.eden.com/~fsg/penguin.html

The most popular examples of mobile stateless entities are located in the field of World-Wide-Web. Mini applications called Applets [Sun95a] were introduced to enhance various aspects of Web browsers (see section 3.4). Applets are typical examples of mobile stateless entities. Web browsers import mobile code and create new instantiations which are non-movable. Because Applets use the Java byte-code representation, we can categorize the code representation as low-level.

A similar example is the Grail-Browser[12] which is able to import code fragments written in Python [van95]. While Java applets are document based, Python applets can also influence some parts of the browser itself.

Other examples are scripts written in the scripting languages like JavaScript, Tcl and SafeTcl.

3.5.4 Security and Mobile Code

One important subject within any discussion about mobile code systems is of course security. Mobile code systems have a natural accumulation of possible security threats. [WJK96b] subdivides possible attacks of mobile code into three parts: Trojan horses, viruses and denial of service.

Trojan Horses

Trojan horses are normally programs that have an official, visible part but also an unofficial, hidden one. The hidden functionality usually has some unwished for effects. Mobile code systems can be misused to implement and spread trojan horses.

Viruses

Viruses are programs that insert copies of themselves into other programs on a host system. Platform independent and mobile code systems are possible invitations for viruses and make them much more dangerous. Platform independent code gives viruses a chance to run (and thus to spread) on most common computer systems and thus a much wider area of spreading.

Denial of Service

When a piece of software starts to use excessive amounts of a system resource, other legitimate requests to the same resource will consequently fail. This sort of attack is known as denial of service (or sometimes as "resource deprivation"). For example a piece of mobile code could start to fill up a disk and thus effect the functioning of other programs.

[12] Grail-Browser: `http://monty.cnri.reston.va.us/grail/`

3.6 Conclusion: Where to Go From Here

In this chapter we listed a few existing approaches and examples within the fields of networking environments, mobile code systems, World-Wide-Web and software composition.

We like to argue here in which direction the remaining chapters of this thesis evolve. Some of the problems addressed are mainly research topics in the Software Composition Group at the University of Berne[13]. Especially solutions in the direction of software composition, for instance composition languages, composition environments and visual composition environments [NT95, MN96].

Other topics discussed like distributed object systems [Mic95, Obj92, Vin93] and mobile agent systems [HCK95, Har95, Hoh95] are already well understood.

The previous discussion about available technologies shows that "mobility" remains as a central property in the field of open, flexible systems. This leads us to requirements like *adaptability*, *heterogeneity*, *security* and of course *mobility* (see section 2.3).

Because we are also interested in the World-Wide-Web domain, we want to solve problems in the direction of open architectures, especially in the direction of open servers and browsers. The Web is an example of a very quickly evolving, distributed system and thus an excellent domain to test and apply new technologies in software development. In addition the Web demands requirements such as platform-independent code, mobile code, updatable code and secure code systems. We want to show examples and new approaches to software architectures, which can extend their run-time system in contrast to well known client/server models. The idea is to show clients, which mobile code abstractions they can import (e.g. related to the applet mechanism) into the their own run-time system in order to fulfill the requirements of flexible open systems.

In chapter 4 we compare four interesting programming languages with capabilities for movable code. We will decide which programming language we should use to implement a prototype. In chapter 5 we define and introduce a basic architecture and the notions "fruitlet" and "run-time framework". Chapter 6 describes the prototype of a "run-time framework". In chapter 7 we describe three examples in the field of open server and open application technology, which uses the technology introduced in chapter 5.

[13] SCG: `http://iamwww.unibe.ch/~scg/`

Chapter 4

Programming Languages for Mobile Programming, a Comparison

This chapter contains a description of various programming languages in the field of mobile programming. It will lead to an evaluation of the language, which we will use to implement a prototype.

We decided to have a closer look at the following programming languages: Obliq, Telescript, Safe-Tcl and Java[1].

Our selection is mainly motivated by the following reasons: all these languages are object-oriented (except Safe-Tcl, see section 4.3) and all the languages are used in projects related to mobile code and mobile agent systems. Thus, our selection includes four of the most known and most mature languages at the moment of writing. In fact, there are a lot of ongoing projects leading to systems and languages that would also possibly fulfill our requirements[2].

In the following we state six criteria, which we expect from a programming language to fulfill our needs. We list the criteria in sequence of priority, higher priorities first.

- OO technology: the language must be based on object-oriented technology. OO technology can stand for software reuse/software design and it is the state of the art software technology. Thus, we see OO as a base technology to build our components.
- Platform independent and mobile code: The language must provide the possibility to write mobile code. Platform independent code is needed in order to overcome the platform islands connected to a network. In contrast to a piece of software that is written for a dedicated platform, a platform independent piece of software can be loaded over a network to be run on any computer with any platform.

[1] a Java, Phantom, Python comparison: `http://www.cgl.uwaterloo.ca/~anicolao/termpaper.html`
[2] see a list of mobile code systems: `http://www.w3.org/pub/WWW/MobileCode/`

- Concurrency: the language should provide concurrency mechanisms. Because modern applications have to interact simultaneously to different events like user interaction and network tasks we would like to have basic concurrency mechanisms in the language. Mobile pieces of software often have to react with networked resources and such tasks can be done most efficiently by using independent threads.
- Security: the language should provide basic mechanisms to create safe components. Because we plan to use our components in networking environments, it is essential to have the possibility to write safe components (e.g. preventing such components from affecting the host computer).
- Communication: the language should provide basic communication libraries for TCP/IP sockets.
- GUI: we need a library/framework which supports access to a graphical user interface from our programming language, because our components should be able to use a graphical user interface. The abstraction of the library/framework has to be platform independent.

The chapter starts with a short description of each language, including discussions related to our criteria.

4.1 Obliq

Obliq[3] was developed by Luca Cardelli [Car95]. Obliq supports object-oriented distributed computation. Obliq is a lexically scoped, untyped, interpreted language. In a lexically scoped language, the binding location of every identifier is determined by simple analysis of the program text surrounding it. Obliq takes the view that identifiers in procedures transmitted over the network are bound to their original locations, even when these locations belong to different network sites. Objects in Obliq are local to a site and do not move. In contrast, network references to objects can be easily transmitted over the network.

Obliq procedures and methods can be freely transmitted over a network. Computations are transmitted using closures instead of source text and lexically scoped free identifiers remain bound to their original site. Thus, an agent without free identifiers is fully disconnected from its original location. In general, a closure consists of a piece of source code and an evaluation stack. Obliq implements closures as pairs consisting of source text and a table of values for free identifiers.

Because of the lexical scoping, transferred computations only have access to data via free identifiers or explicitly received procedure parameters.

Although Obliq is an untyped language (no static type checking), the Obliq run-time is strongly typed.

[3] Obliq: `http://www.research.digital.com/SRC/Oblique/Oblique.html`

In Obliq a thread is a virtual sequential instruction process. A thread may stop execution on one site and continue execution on another site. Multiple threads may be executed concurrently.

Obliq is object-based according to the classification of Wegner [Weg87]. Objects can be constructed directly or cloned by other objects (prototype approach). All methods as well as value fields are embedded in the object itself and thus suitable for networking purposes.

4.1.1 Visual Obliq

Visual Obliq is an environment for programming and running distributed multi-user GUI applications. The interface builder is a visual composition environment, similar to the Visual Basic environment. It outputs code in the Obliq language. A Safe Visual Obliq Interpreter is a special purpose interpreter used to interpret the applications that come over the network. The interpreter is considered safe, because it screens all unsafe operations based on their arguments. A user specific configuration file contains regular expressions that specify which operations are to be allowed and which are to be locked. One interesting work using Visual Obliq in the field of agents is described in[KL96].

4.1.2 Discussion

Distributed lexical scoping allows for flexible distributed computation and transferred computations behave correctly. However, the flexibility can result in hard to estimate and undesirable network traffic. The current Obliq implementation supports access to many Modula-3 libraries. Unfortunately, Visual Obliq supports the GUI using the Trestle toolkit, thus the abstractions of the GUI classes are not platform independent. But Obliq/Visual Obliq satisfy all our requirements.

4.2 Telescript

Telescript[4] is a technology to integrate computers and networks that link them [Gen95]. The Telescript environment helps to implement active, distributed network applications. The Telescript model assumes an electronic world, which consists of places, each occupied by mobile agents. Mobile agents are able to travel from place to place. The authority of places and agents is represented by a telename. Mobile agents and places can neither withhold nor falsify these telenames. Each place is associated with a teleaddress.

In general, an agent is a piece of code which travels along different places to meet other agents. Rather than using remote procedure calls to interact, agents use remote programming to interact with remote places. This can improve the performance of interactions.

[4] Telescript: `http://www.genmagic.com/Telescript/index.html`

4.2.1 Language Characteristics

- Safety: an agent is not allowed to inflict damage to the host computer
- Portability: an agent or place can be executed on different platforms
- Exdendability
- Elevation: the language makes no distinction between volatile and non volatile storage.

The Telescript language is an object-oriented remote language. Telescript uses interpreters to execute pre-compiled code. A Telescript engine allows execution of the language's object programs. Telescript knows about different API interfaces to the resources of the system (e.g. storage, communication, access to external applications). The External Application Framework of Telescript will phase out in the next major release of Telescript and will be replaced by a different mechanism that provides greater security.

4.2.2 Language Concepts

Objects in Telescript can be passive (e.g. String, Boolean) or active (e.g. an agent or a place). An object represents both information and information processing. An object has an externally invisible implementation and an externally visible interface. The interface consists of operations and attributes. Operations are tasks that an object (responder) performs on a request from another object (requester). The requester and the responder may be the same object. A requester provides one or more objects as the operation's arguments to the responder. An operation can fail or succeed. If it succeeds the responder can return a single object as a result to the requester. If the operation fails, the responder throws a single object (exception) to the requester. Every operation has an interface and an implementation. The interface reflects the signature of an operation.

An attribute is itself an object. An attribute can be set or read at a another's request. A normal attribute is the product of two operations, called setter and getter. If an attribute is read only it only has a getter operation.

The state of an object is represented by zero or more objects, its properties.

Constraints are used to dictate the type of an object and the types of arguments in operations. These constraints are statically checked by the compiler and dynamically by the engine (interpreter).

In Telescript, every object is owned by an agent or a place. Agents and places own themselves. An object can modify other objects, that the current owner of this object owns.

In Telescript a class is an object itself. A class specifies its own interfaces and implementation as well as those of its instances. As in other OO languages a class determines its instances.

The class family concept of Telescript allows for parameterized classes (templates).

Telescript has more complex inheritance mechanisms than other OO languages. Two different kinds of classes are known: mix-ins and flavors. A mix-in class can not have instances. Every immediate subclass of a mix-in is a subclass of a class the mix-in designates as its anchor. The ancestor or its anchor is a flavor. Interface and implementation inheritance is defined in the language.

4.2.3 Class Libraries

The predefined Telescript classes are based on the following concept. Major abstractions are used for:

- Places: places represent meeting points for agents. Places help agents to interact.
- Agents: a basic unit, which is able to travel from place to place.
- Processes: an abstraction for places and agents.
- Permits: an abstraction of process permissions.
- Patterns: analyzing and modifying tools for strings.
- Calendar: analyzing and modifying tools for times.

There are no abstractions for graphical user interfaces (GUI) or widgets available. Basically, it is possible to call external functions (e.g. C++) and therefore use external libraries to access the graphical user interface. Doing so, we will lose the feature of platform independent code and new security questions will appear.

4.2.4 Discussion

The programming language reflects object-oriented technology. Pre-compiled code enables execution on different platforms (tele-engine) and a process abstraction to write concurrent programs is also available. The language concepts (unfortunately) seem to be quite complicated. The lack of platform independent abstractions for communication (TCP/IP) on one hand and the lack of abstractions for graphical user interfaces on the other hand are major drawbacks of Telescript at the moment.

4.3 Safe-Tcl/Incr Tcl

The Safe-Tcl language was developed by Nathaniel Borenstein [Bor92, BR93] as a derivation of the Tcl (Tool Command Language) scripting language. The application domain of Safe-Tcl is active mail, which we can classify as stateful mobile components (see section 3.5). The syntax of Safe-Tcl is identical to the syntax of Tcl. The author describes Safe-Tcl as an "extended subset" of Tcl. Although Tcl is not an object-oriented language (and thus Safe-Tcl is not object-oriented), we discuss Safe-Tcl here, for two reasons. First,

there is an object-oriented version of Tcl, called incr Tcl, and second, Safe-Tcl addresses some basic problems about security in the fields of active mails and mobile components.

The major concept of Safe-Tcl is to remove all dangerous primitive functions/procedures of the Tcl language to obtain a safe language. Dangerous functions/procedures means everything which is able to destroy the integrity of the host computer (e.g. read/write from/to the disk, read/write from/to the network). In addition Safe-Tcl adds certain new primitives to the restricted Tcl. A Safe-Tcl program will always use two interpreters to execute. An unrestricted (trusted) full Tcl interpreter and a restricted (untrusted) interpreter for Safe-Tcl. Untrusted programs only have access to the trusted interpreter via a mechanism defined in Safe-Tcl language.

Tcl is an interpreted scripting language with a strong emphasis on strings. Tk is the most commonly used extension to Tcl. It represents easy to use abstractions of graphical user interface components usable with Tcl. Tk is also the major reason for the current popularity of Tcl (often referred to as Tcl/Tk). Tk helps to cut down the development time of an application. Another popular extension to Tcl is DP (distributed programming), which helps to write client/server applications in an easy fashion[5].

Both, Tcl and incr Tcl have an extra language extension (incr Tcl and incr Tk), which helps to build large applications with Tcl/Tk using object oriented technology.

4.3.1 Discussion

Safe-Tcl is not really object-oriented. That means that we would need to modify the incr Tcl language towards a "Safe-incr-Tcl" language. Tcl interpreters are available on UNIX, Macintosh, Windows and NT platforms. Tcl supports no built-in concurrency constructs. Unfortunately Tk and incr Tk, which helps to access the graphical user interface are based on Motif widgets and thus not platform independent.

4.4 Java

The object-oriented programming language Java was recently developed by Sun Microsystem[Sun95a, GM95]. Java is in many way related to C++, but the designers removed many unsafe constructs of C++ (e.g. there are no pointers in Java). Sun uses pre-compiled code (Java byte-code) to reach the goal of platform independent code. The pre-compiled Java code usually runs on an interpreter (Just-In-Time compilers, which compile byte-code to machine code, now becoming available). Currently there are interpreters available for all common platforms (Solaris, Linux, Microsoft NT, Mircosoft Windows 95, MacIntosh). Java was designed to improve network based information environments, like the World-Wide-Web. In order to implement concurrent programs, Java provides a multi-threaded programming environment. The Java language does not support mechanisms to move objects directly within a networking environment, but Java allows the loading of new classes during run-time. New classes can either be loaded from a local machine or from

[5]DP: `http://ftp.aud.alcatel.com/tcl/extensions/tcl-dp3.2.README`

any other remote machine in the network.

4.4.1 Object-Oriented Properties

Java is class based and every class is a type. Every object is instantiated from a class. In contrast to C++, Java also has a notion of interfaces and these interfaces are also types. An interface is a set of function signatures. Regardless of the class hierarchy, any arbitrary class can implement any number of interfaces. A class can inherit from another class (single inheritance). Interfaces can only inherit from other interfaces. The Java run-time keeps a lot of meta-information about classes, objects and inheritance. For example an object is able to determine its ancestors or its interfaces.

4.4.2 Security

The Java run-time system loads every class (even built-in classes) either from the local file system or from the network. The first of several security levels is to type-check all classes after the loading step. In fact, Java passes the byte-code through a simple theorem prover, which ensures a safe execution of the byte-code (e.g. no interpreter crash, no unsafe operations). The possibility of handling network-loaded code differently from code loaded from the local file system helps to check imported code. For example, it is possible to lock basic I/O facilities (or at least do some monitoring) for classes loaded from a remote side.

4.4.3 Class Libraries

Java provides a standard set of classes structured into packages to extend the basic language facilities. All these packages reflect abstractions in a platform independent manner. One interesting package is the AWT (abstract windowing toolkit) package. AWT is a completely platform-independent framework in order to implement graphical user interfaces.

4.4.4 Discussion

Java fulfills all our requirements. The main difference to the other languages is the absence of a real object moving mechanism. Although this is not impossible [Hoh95], we have no direct language constructs which supports mobile agents. Java enables some basic mechanisms in order to implement security issues (see appendix B).

4.5 Conclusion

It seems that Java best fits our needs. Because we do not want to design another agent package, the lack of really movable objects does not carry too much weight. Java is very popular at the moment and thus there are a lot of running projects related to the Internet, World-Wide-Web and mobile agents. Projects like PJava (Glasgow Persistent

Java) [AJDS96] or Active Objects (Doug Lea)[6] will help to improve the language. Java is already strong in the sense of platform neutrality. Interpreters for most of the common platforms are available. Concurrency constructs are integrated into the language itself, which makes concurrent programming very easy and natural. Major design decisions related to security provide the basics to write safe, open applications. First of all, the absence of pointers and pointer arithmetic in the programming language itself helps to prevent coding mistakes and provide basic security. The Java compiler also prevents illegal cast operations. Furthermore, the interpreter checks every class before execution to ensure that the code plays by the rules [Yel96]. The object-oriented standard libraries provide good abstractions to write platform independent code. Especially the AWT framework and the abstraction of basic TCP/IP communication mechanisms, are very suitable. The language itself is very simple and clean.

Telescript is a very interesting language, and it is very suitable for any agent-like remote programming task. The major drawback is the lack of a platform independent abstraction to access the graphical user interface. At the time of writing, Telescript has not gained wide popularity in the Internet community.

Although the lexical scoping approach of Obliq is interesting, it leads to a different view of distributed/mobile components. In Obliq the basic cells of distribution are identifiers, which are somehow "too small".

Safe-Tcl is not really object-oriented at the moment. The related Tk extension of Tcl is Motif based. The procedural approach would prevent the use of object-oriented base technology.

[6]Active Objects: `http://g.oswego.edu/dl/pats/aopintro.html`

Chapter 5

Model

This chapter describes the model and gives some details of our "component understanding" which we use in this thesis. The central point of this chapter is the definition of mobile components and fruitlets. Furthermore, the description of the relationship between fruitlets and the run-time framework is central. The run-time framework is considered as the basic shell to import fruitlets and as a provider of basic services needed by fruitlets.

The first section defines the notions of "Component", "Mobile Component" and the second section is about the terms "Fruitlet" and "Run-time Framework".

5.1 Definitions

This section provides some definitions that we use in this thesis. Whenever possible we rely on available definitions in the context of software composition. The main question is to find definitions for component and mobile component.

5.1.1 Component

The term "Component" already has very wide application in computer science. On the one hand, components can be single software applications within a whole application package, but on the other hand components can also be objects within their run-time environment. In computer science, we usually use the notion component, if any abstraction of software can be referred to as an encapsulated entity (and to some degree stand alone) which is able to live within a bigger environment (but not necessarily).

Especially in fields like open software applications, distributed applications and of course software composition, people increasingly use the notion of component (although usually with slightly different meanings).

Thus, it is very hard to find corresponding definitions of the important term "Component". We refer in this thesis to a basic definition in [ND95]. The paper focuses on software components and declares a component as a "static abstraction with plugs". Where "static"

means a long-lived entity that can be stored independently and "abstraction" means a more or less arbitrary boundary around a piece of software. "Plugs" can be any well-defined ways to communicate and interact with the component, in short any mechanism which helps to use or reuse the component.

5.1.2 Mobile Component

The notion "mobile component" is applicable to many common software entities. For instance, a mobile software agent can be referred to as a "mobile component". In the same way we can apply the notion "mobile component" to an Applet (see section 3.5).

We can say that a mobile component is a component where the encapsulated software abstraction is transportable. Mobile components can be *stateful* or *stateless* software entities (see section 3.5). The code representation within the software abstraction may be "low-level" or "high-level" (see section 3.5.1).

5.2 Proposed Model

In chapter 2 and chapter 3 we uncovered and described some requirements of open, flexible systems within distributed networking environments.

Starting with the Applet approach in mind (see section 3.4), we want to build a more general software architecture with stateless mobile components as building blocks. Applets are used to build animated Web pages and they can also be used to develop whole applications within a browser environment. From their very design, Applets are always "visual" during their lifetime within a Web browser. In short, Applets require a fixed protocol also including methods for graphical operations and thus they are not always general enough. In contrast, our approach does neither impose a graphical representation nor a Web browser environment.

In this section we explain a small model in order to outline a general structure of stateless mobile building blocks.

5.2.1 Fruitlet

Open network applications could consist of stateful and stateless entities (see section 3.5). This thesis wants to deal with non-movable instantiations of mobile components. Thus, we introduce at this point the notion of a "fruitlet"[1].

Usually the boundary of a component in general is no longer visible during run-time. Because we want to attain some degree of adaptability during system's run-time, we require knowledge about the boundary of "fruitlets" during run-time (e.g. in order to load, delete, update, recompose fruitlets).

[1] With thanks to Theo Dirk Meijler for the invention

We can also refer to "fruitlets" as a sort of template in order to create fruitlets instantiations (e.g. a set of objects).

We define properties of fruitlets as follows:

- Fruitlets are mobile components with "stateless software abstraction" (thus the instantiations of fruitlets may be non-movable, see section 3.5.3).
- Fruitlets are "platform mobile" and "location mobile" (e.g. by using a low-level platform independent code representation, see section 3.5).
- Fruitlets can ask the run-time framework to create new non-movable instantiations by providing the address of a fruitlet.
- Fruitlets provide their functionality through one or more interfaces. No other entry points to a fruitlet are available.
- Every fruitlet provide a well-known unique interface, which allows for basic queries to a fruitlet (e.g. accessing fruitlet parameter, checking implemented interfaces).
- After the importing step, predefined parameters of a fruitlet can be set using the well-known interface (e.g. by specifying addresses of other fruitlets).
- Fruitlets have to be "security checkable" (i.e. the surrounding run-time framework must control all the accesses of fruitlets to system resources and system services).

"Stateless software abstraction" means that the mobile part of fruitlets must not carry any state (see section 3.5.3).

Because fruitlets are stateless, it is sufficient to use platform independent code representation to gain "platform mobility" and "location mobility". Platform independent means that code fragments behave equally, independent of hardware, operating system and graphical user interface.

By "security checkable" we mean that each fruitlet contains mechanisms to fit into a particular security concept. The reason is that clients of fruitlets can apply a security policy to ensure the integrity of the component itself, of other components and of the system (see section 6.2.4). The security concept protects various system resources (e.g. file system, network) from uncontrolled access by fruitlets.

Thus, fruitlets must live on a software layer on the top of a specific platform in order to fulfill all the requirements. Such a layer has to ensure a unique "software interface" to different platforms.

5.2.2 Plugs of Fruitlets

We use a set of function signatures (interfaces) in order to communicate with fruitlets (see also section 3.2 and section 3.3). An interface in this sense is a strongly typed contract between software entities. An interface is also an articulation of expected behavior and responsibilities of software entities.

5.2.3 Naming and Locating Fruitlets

We define the "address" of a fruitlet as an URL, which consists of a host part and a type part to locate a specific fruitlet on the Internet. For instance, `http://foo.unibe.ch:7000` reflects the host part and `/unibe/omo/Omo` the type part in the fruitlet address `http://foo.unibe.ch:7000/unibe/omo/Omo`. The same fruitlet can have several addresses, for instance if it resides on two different hosts. The run-time framework makes a clear distinction between different addresses.

5.2.4 Run-Time Framework

We use a basic software environment on top of a specific platform which provides basic mechanisms to integrate and run new fruitlets. We call such a basic software environment "run-time framework". Basically, a run-time framework is responsible for instantiation of new fruitlets during run-time. It should also provide concrete transport mechanisms to import fruitlets over the network. In order to check for security constraints a run-time framework must provide a security concept.

We can summarize the properties of a run-time framework:

- A run-time framework is able to instantiate fruitlets, by using the address of a fruitlet.
- A run-time framework checks all system accesses of imported fruitlets during run-time in order to guarantee the system integrity.
- A run-time framework collects information about fruitlets (e.g. which interfaces a specific fruitlet implements).
- A run-time framework instantiates one (parameterizable) default fruitlet at startup-time.

5.2.5 Security Impacts

Because software reuse was mainly based on a white-box reuse scheme, security did not concern the software developer directly. Security issues were mostly relegated to the underlying operating system. If we introduce systems and mechanisms for black-box reuse (black-box frameworks, component frameworks) and especially mobile code and mobile components, we should start to think about security as an essential point of software development. If we dream of robust components, developed by different vendors and assembled by programmers or users, we have to build mechanisms, which help to take care of security. We think that component security should be introduced as a general topic into the context of software composition. Any component framework or component system, which is able to load or to use/reuse components has to provide some basic security features. Otherwise the risk of integrating dangerous components into our own

applications, especially components from untrusted vendors or developers, will be very high.

In our model we provide a solution, which restricts access to dangerous resources of a computer system a given component (fruitlet) can have (see also section 3.5.4).

In chapter 6 we introduce a concrete security concept for fruitlets (see section 6.2.4).

Figure 5.1 gives an overview of the principle of our Fruitlet approach.

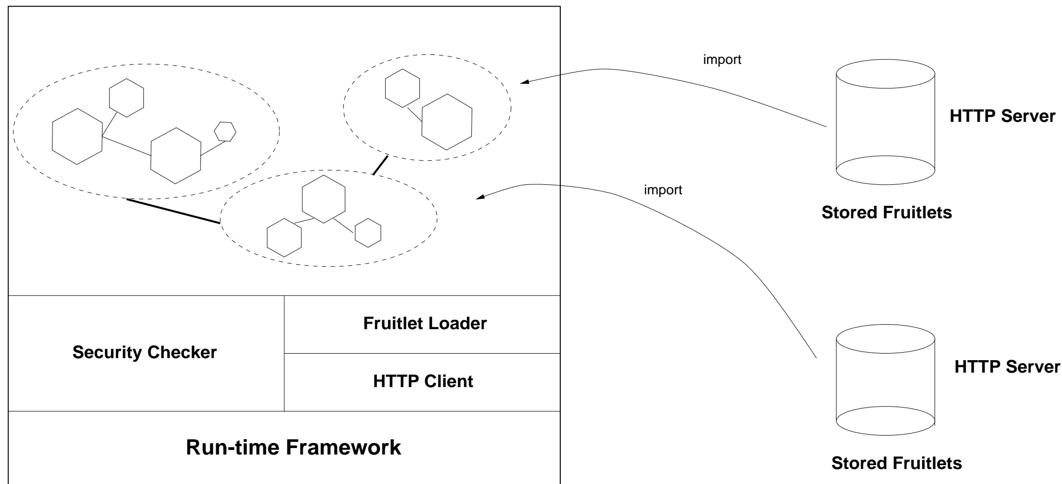

Figure 5.1: Overview of the Run-time Framework

5.3 Conclusion

Within the description of a model we discussed notions like "Component" and "Mobile Component". We gave definitions of our meanings of "Fruitlet" and "Run-time Framework". On the one hand "Fruitlets" are conceptually higher-level than other concepts like movable classes (e.g. Java classes). They include, for instance, a security concept as well as basic parameterization mechanisms. On the other hand "Fruitlets" are more general than the related Applet approach.

Fruitlets lead us to an approach where applications consists of distributed code fragments. During run-time the instantiations of these mobile building blocks form the local run-time structure of an application. Thus, fruitlets help to improve flexibility and openness of a system and also help to maintain the evolution of an application (e.g. adaptation of systems to new requirements and "automatic" code upgrades).

The properties of fruitlets also help to attain requirements like mobility, heterogeneity, adaptability and security of networking based open applications (see also section 2.3).

Chapter 6

Prototype

This chapter discusses and describes the design and implementation details of the basic architecture.

With respect to the results of chapter 4, we decided to use the Java technology and the Java programming language from Sun Microsystems [Sun95a] to implement the prototype. The Java technology best fits our needs for platform neutrality, security, communication, concurrency mechanisms and also provides solid OO-technology.

With Java in mind we can give a more concrete idea of the notions “fruitlet” and “run-time framework” in the next section.

6.1 The Prototype meaning of “Fruitlet” and “Run-Time Framework”

The central notions of chapter 5 (fruitlet, run-time framework) lead us to most of the design decisions within this chapter. We give more precise meanings of “Fruitlet” and “Run-Time Framework” concerning the concrete implementation of our prototype.

As a basic decision, we will use the Java byte-code to describe the software abstractions of fruitlets. A fruitlet contains one or more classes written in Java byte-code. We consider this decision to be sufficient to fulfill the requirement of platform independence abstractions of fruitlets (surely sufficient for a prototype implementation). The Java byte-code [Sun95b] has also some basic properties which can be used to fulfill the requirement of “security checkable” (see appendix B). It allows us to implement a security concept into the run-time framework of the prototype [WJK96b].

Java allows for a notion of interfaces (a set of function signatures), which we use in order to communicate between fruitlets and the run-time framework. At the moment we regard the interface terminology as sufficient in order to articulate expected behavior of fruitlets.

Therefore, we can propose a first approach concerning the run-time framework. The run-time framework consists of a Java byte-code interpreter as well as some additional Java

byte-code, which runs on the interpreter and helps to guarantee security issues and helps to fulfill the basic properties of the run-time framework. Amongst other things, we are forced to integrate a loader for fruitlets into the run-time framework. Because Java interpreters allow us to introduce new class loaders, this should be possible.

In general, we prefer to use the Java programming language in order to create the necessary Java byte-code. This decision is not fundamental to our concept but is the most convenient way at the moment. For an alternative see "The Ada 95 Experiment" in section 7.3.3.

6.2 Design

This section describes the design and implementation details of the prototype. The subsections explain the prototype's solution of various points like security, component loader and class loader. First of all we introduce the two most important interfaces of the design. Firstly, the interface `ManagerIfc`, which reflects the interface to the component manager of the run-time framework. Secondly, the interface `ComponentIfc`, which is the generic interface that all fruitlets must provide to the public. Both interfaces are located in the package `unibe.interfaces`.

6.2.1 ManagerIfc

The component manager is a central part of the run-time framework. It is responsible for loading fruitlets and creating new instances. In fact, it is the only part of the run-time framework which is allowed to load and create new instances of fruitlets. The manager also enables various security checks in cooperation with the class loader of the run-time framework (see section 6.2.4).

Basically the `ManagerIfc` provides methods of creating new instances of a fruitlet. The fruitlet is specified by its address. After a request for a new instance appears, the manager searches the specified fruitlet in a hashtable and decides if the fruitlet is already available. If the specified fruitlet is not available it forwards the request to the `WebLoader`. The `WebLoader` then searches for the fruitlet on the Internet using the specified address. Both methods return a reference to the created instance. The reference is of type `ComponentIfc`.

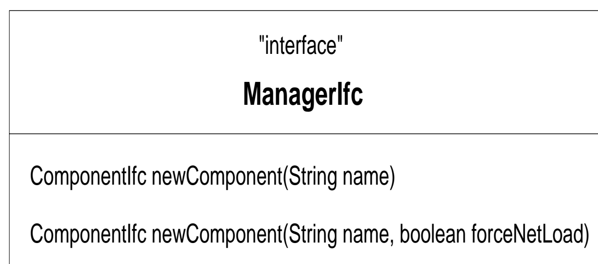

Figure 6.1: Interface ManagerIfc

The program listing of the interface in Java:

```
package unibe.interfaces;

/**
 * This interface enables access to the manager in order to
 * load and instantiate new fruitlets.
 */
public interface ManagerIfc {

    /**
     * Creates a new instance of a fruitlet.
     *
     * @param name          the address of the fruitlet
     */
    ComponentIfc newComponent(String name);

    /**
     * Creates a new instance of a fruitlet. If forceNetLoad is
     * true, the fruitlet will always be loaded from the Internet.
     *
     * @param name          the address of the fruitlet
     * @param forceNetLoad  true, in order to force loading from Internet
     */
    ComponentIfc newComponent(String name, boolean forceNetLoad);
}
```

The only class in the run-time framework, which implements the interface `ManagerIfc` is `Manager` (see section 6.2.5).

6.2.2 ComponentIfc

The `ComponentIfc` is the central interface of any fruitlet. Every fruitlet of the prototype has to implement at least this interface. The interface provides a set of methods we can expect from every fruitlet. `ComponentIfc` provides a minimal set of methods in the prototype. For future usage we have to extend this interface with more functionality (e.g. access to meta information of a fruitlet, see chapter 8).

The methods `parameters`, `getParameter`, `setParameter` can be used to manage predefined slots of a fruitlet (see section 6.2.3).

The program listing of the interface in Java:

```
package unibe.interfaces;
import  java.lang.String;
import  java.util.Enumeration;

/**
 * This interface is the basic access plug of any fruitlet.
 * Each fruitlet has to implement this interface.
 */
public interface ComponentIfc {
```

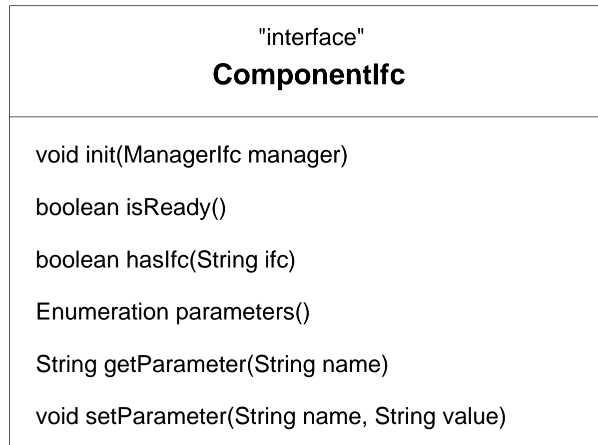

Figure 6.2: Interface ComponentIfc

```
/**
 * The run-time framework starts this method after the creation step
 * of an new instance.
 *
 * @param manager     the manager of the run-time framework
 */
void        init(ManagerIfc manager);

/**
 * Returns true, if the instance is ready.
 * (e.g. all predefined parameters are set)
 */
boolean     isReady();

/**
 * Returns true, if the fruitlet implements interface ifc.
 *
 * @param ifc         the interface name to check
 */
boolean     hasIfc(String ifc);

/**
 * Returns an enumeration of all settable parameters
 */
Enumeration parameters();

/**
 * Returns the value of parameter name.
 *
 * @param name        the name of the parameter
 */
String      getParameter(String name);
```

```
    /**
     * Sets the parameter name to a new value.
     *
     * @param name        the name of the parameter
     * @param value       the value of the parameter
     */
    void        setParameter(String name, String value);
}
```

6.2.3 Fruitlet Parameterization

Parameterization is a general composition mechanism. We introduce a general parameterization mechanism into our prototype understanding of fruitlets. Often, a fruitlet A will use functionality of another fruitlet B in a particular design an thus the main purpose of fruitlet parameters is to set addresses of other fruitlets within a given fruitlet (e.g. fruitlet A has the address of fruitlet B as a parameter).

In order to parameterize a fruitlet, we can define a set of *slots* for each fruitlet. At the moment a slot has a name and a value. In the prototype the name and the value must always be of the type string. A fruitlet can define default values for each slot.

There are two ways to set slots of a fruitlet. Firstly, during the instantiation phase of a fruitlet the component loader looks for a text file with the extension `.parameter` at the same location as the fruitlet itself (for instance we use the address `http://idefix.isburg.ch:7000/unibe/foo.parameter`). The file can contain line oriented pairs `<name> = <value>`. The component loader sets these slots (if the corresponding fruitlet recognizes such parameters) during the instantiation phase of a fruitlet. Secondly, after the instantiation a client of a fruitlet's instance can use the `ComponentIfc` to manipulate the slots.

6.2.4 Security

Mobile components introduce a central security problem. Imported components could damage the integrity of the host computer by uncontrolled use of the local services. In order to ensure a secure environment, we have to control and restrict the access and use of locally available resources and services on a computer system (see also section 3.5.4).

The Java language in cooperation with the Java packages leads to a basic environment to write secure code. The Java technology can be referred to as an enabling technology to build secure systems for mobile code (Java should in fact do that, see [DFW96]).

Each standalone Java program can specify and implement a "security manager". Every method in the Java packages which could damage the system integrity, calls a corresponding method in the system security manager, before executing any dangerous system calls. The security manager offers some methods to determine the current execution environment and can therefore decide to stop the execution. The security manager can evaluate the class loader which is responsible for imported code. More details about Java's security concept can be found in appendix B.

We use the mechanism of the security manager and class loaders in our prototype to implement security policies.

The system wide security manager which we introduce in the run-time framework is called `GeneralSecurityManager`. It dispatches all calls related to security constraints to the responsible class loader. Every class loader has an instance of `ComponentSecurityManager` which knows about the security policy of a specific fruitlet. The component loader (`Manager`) of the run-time framework creates a new class loader for every imported fruitlet type. In the prototype implementation all class loaders are of the type `WebLoader`. There are several ways to authorize a fruitlet (see also [WJK96b]) that wants to use resources and services. The design of the prototype allows the use of the type and the source (the address) of a fruitlet to specify the security policy of a fruitlet. Thus, we can specify different security policies for each fruitlet by using the fruitlet's address.

This leads us to an overview of the class diagram:

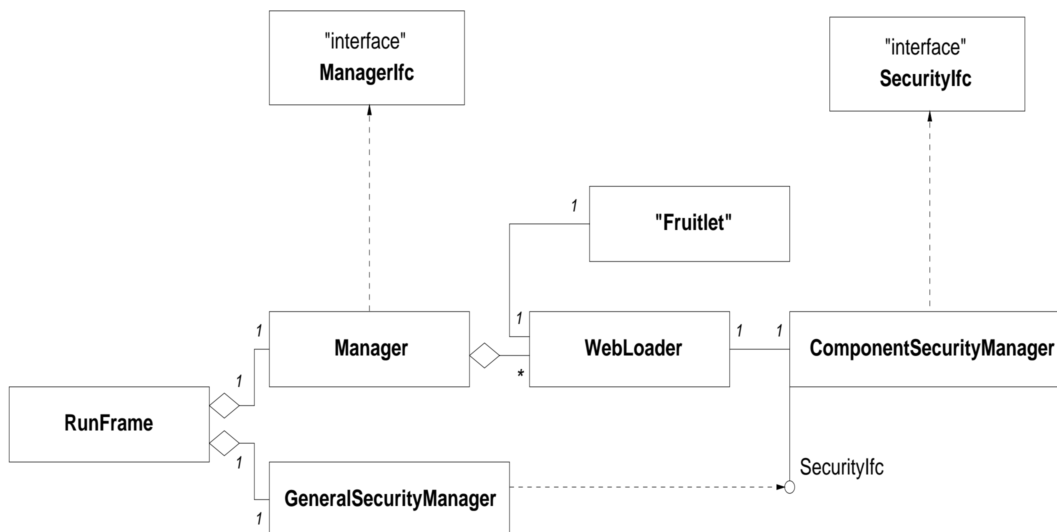

Figure 6.3: Class Diagram: Overview Run-Time Framework

Fruitlet Security

We decided to implement the following security concept in our prototype. Each fruitlet has its own security policy. The security issues of a specific fruitlet are controlled by the run-time framework. Thus, we can specify the security policies on the local system. In the prototype implementation security constraints of fruitlets are stored in a text file. The name of the file is constructed by the address of a fruitlet. For instance, if the address is `http://idefix.isburg.ch:7000/unibe/foo` we use the file name `<SecurityDirectory>idefix.isburg.ch.7000.unibe.foo.security`. The directory of all security files (`<SecurityDirectory>`) is a parameter of the run-time framework (specified in `RunFrame.txt`).

The responsible class loader tries to find its security file (using the address of the fruitlet).

If no security file is present, the fruitlet has no access to the controlled resources.

The basic idea is, that a provider of a fruitlet can distribute a suitable `.security` file. The provider must then justify the opening of files and the use of network connections. The administrator of the local site can then decide to install the `.security` file or not.

We restrict ourselves to control only the following system resources in the current prototype:

- Disk access
- Network access
- Possibility to shut down the run-time framework

We never allow execution of other programs from within a fruitlet. All graphical windows created by fruitlets will be decorated with a warning string (`RunFrame: Fruitlet Window`). Other access controls (e.g. access to running threads) could be implemented in a similar way.

If the run-time framework does not find a security file for a particular fruitlet, all accesses are restricted.

Security Files

The `.security` files contains line oriented entries. Each entry has a keyword and a value. The keyword specifies the access mechanism and the value declares the resource. The syntax of the `.security` file is:

- `FileRead=<filename>`: this entry enables the fruitlet to read the file <filename>. If <filename> ends with a `*`, every file starting with <filename> will match.
- `FileWrite=<filename>`: this entry enables the fruitlet to write to the file <filename>. If <filename> ends with a `*`, every file starting with <filename> will match.
- `Connection=<host>,<port>`: this entry enables the fruitlet to open a network connection to the specified connection end point.
- `ServerPort=<port>`: this entry enables the fruitlet to create a listening port (server port).
- `Exit=<status>`: this entry enables the fruitlet to shut down the run-time framework with status code <status>.

Example of a `.security` file:

```
FileRead     = /home/gertsch/*
FileWrite    = /home/gertsch/temp/*
Connection   = server1.isburg.ch,110
Connection   = 193.5.168.13,110
```

6.2.5 Component Loader

This section describes the design of the component loader. The component loader is a central feature of the run-time framework. The run-time framework has exactly one instance of the component loader. The component loader creates a new class loader (type `WebLoader`) for each new fruitlet type. The system wide security manager (type `GeneralSecurityManager`) ensures, that only the component loader is allowed to create new class loaders. Thus, there is no possibility for imported fruitlets to create new class loaders. The class retriever mechanism of the class loaders is based on the HTTP 1.0 protocol. This means that all HTTP servers are able to distribute fruitlets. The `WebLoader` class implements a restricted HTTP 1.0 client in order to retrieve `.class` Java byte-code files and `.parameter` text files (see section 6.3.2).

The private class loader object of every fruitlet implements the security policy of a specific fruitlet. Because all related classes of a fruitlet also refer to the private class loader object (see "A closer look at Java" in appendix B) the loader can control all access requests to guarded resources and services of the fruitlet (e.g. disk access, network access) and all related classes it has loaded (see section 6.2.4).

Class Cache

Our prototype `WebLoader` maintains a code cache for all imported classes. Firstly, every instance of `WebLoader` maintains a class cache in the memory and secondly the instances of `WebLoader` maintain a class cache on the disk. This means they use the "if-modified-since" mechanism of the HTTP/1.0 protocol if a requested class already exists in the disk cache. The cache directory is a parameter of the run-time framework (specified in `RunFrame.txt`). The name of the file in the cache is generated using the address of a class. For instance, if the address is `http://idefix.isburg.ch:7000/unibe/foo` we use the file name `idefix.isburg.ch.7000.unibe.foo.class`.

6.3 Prototype Packages

This section contains a collection of class diagrams. We show the important design details of the different packages which belong to the core implementation of the run-time framework.

More detailed information about the implementation of the classes is available in HTML format.

6.3.1 Class Diagram of Package `unibe.componentloader`

This package contains the implementation of the component loader (see section 6.2.5). Figure 6.4 shows the class diagram of the component loader package.

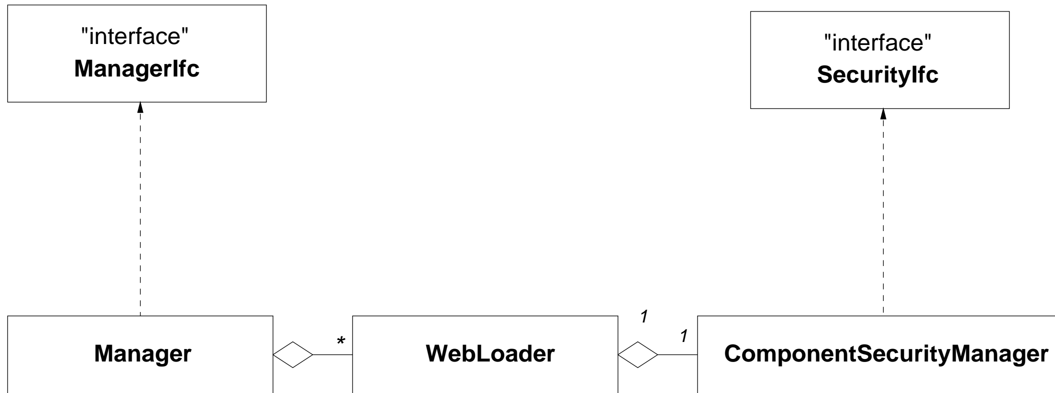

Figure 6.4: Class Diagram: Component Loader

6.3.2 Class Diagram of Package `unibe.net.http`

The package `unibe.net.http` contains a restricted HTTP/1.0 client. The `WebLoader` class uses this client implementation to fetch the `.class` and `.parameter` files from the Internet. As an overview we provide here a class diagram of the main classes in the package.

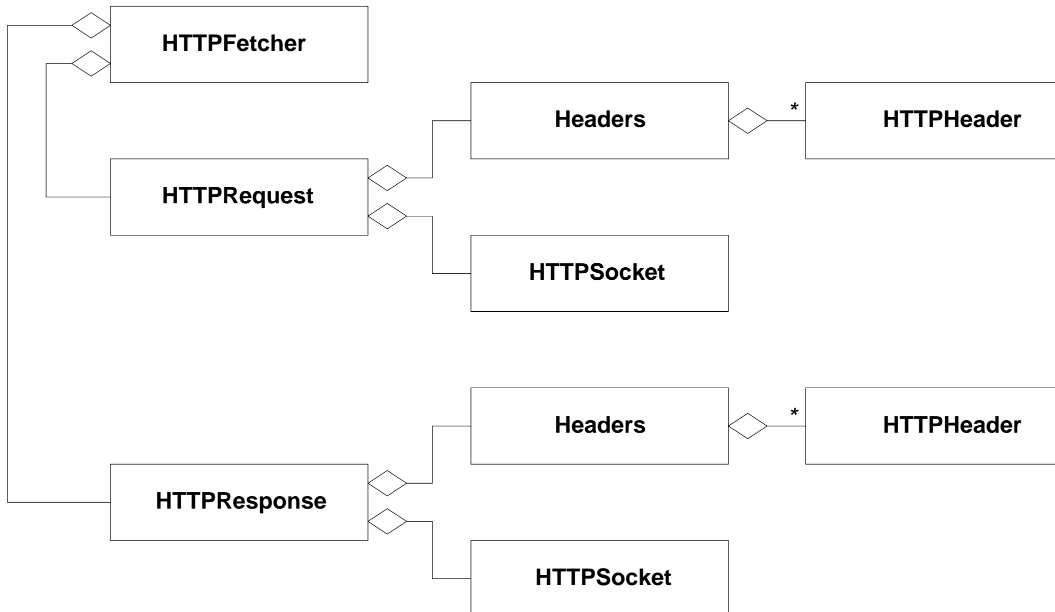

Figure 6.5: Class Diagram: HTTPFetcher

6.3.3 Package unibe.security

The package `unibe.security` contains the security manager of the run-time framework (`GeneralSecurityManager`). It also contains the `ComponentSecurityManager` which is responsible for the security policy of a particular fruitlet (see figure 6.3).

6.3.4 Class Diagram of Package unibe.run

In package `unibe.run` we find the core of the prototype. The class `RunFrame` reads the parameters `SecurityDirectory` and `CacheDirectory` from the file `RunFrame.txt`. Furthermore, it instantiates a `Manager` and a `GeneralSecurityManager`. The last step is to load the fruitlet provided on the command line.

For example: `% java unibe.run.RunFrame http://idefix.unibe.ch/unibe/foo`

Figure 6.6 shows the basic class structure.

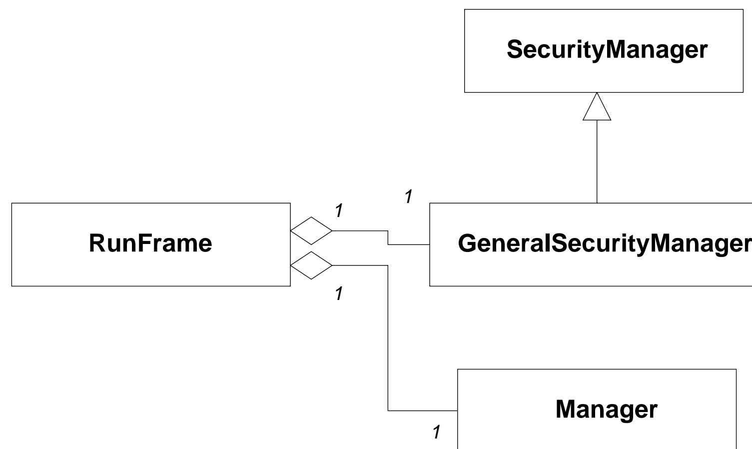

Figure 6.6: Class Diagram: RunFrame

Chapter 7

Use Cases

This chapter introduces three examples in which we use the approach of fruitlets.

This chapter should help to verify the approach of our model (see chapter 5) and also test the basic functionality of the prototype.

In this chapter we discuss the design of the example on a relatively high level. For a more detailed description of the design we refer to appendix A.

As a first example, we explain a general server fruitlet. This example shows a general server architecture to build TCP/IP based Internet servers [Com95, CS93]. In example two, we show a prototype of an open HTTP server, which uses the basic architecture of example one.

With example three we introduce an open message system. This architecture shows an open message organizer in order to integrate different common message systems.

All examples use the basic architecture of the prototype (see chapter 6).

7.1 General Internet Server

Servers are one kind of generic software entities on the Internet. Internet servers provide a basic architecture for most of the well known Internet services, like electronic mail, World-Wide-Web and file transfer, because all the responsible protocols are based on the client/server paradigm on top of the TCP/IP protocol.

The basic architecture of a TCP/IP based server is quite simple and remains the same for most of the common protocols [CS93]. Usually, a server is listening on a socket with a well known port number in order to accept incoming calls. Once a call has been accepted, the server starts a new task with the communication channel (usually a socket) as a parameter.

7.1.1 Design

We introduce a fruitlet `GenericServer` included in the package `unibe.internet` to implement a generic server architecture. `GenericServer` has two slots (see section 6.2.3). Firstly, a slot with name "Port" and secondly a slot with the name "RequestServiceComponent". The "Port" slot specifies the port on which our server should listen and the "RequestServiceComponent" slot specifies the "helper" fruitlet, which performs an incoming request. This fruitlet has to implement the interface `ServiceRequestIfc`. The `GenericServer` itself implements `ServerIfc` (see appendix A).

For each incoming request the `GenericServer` fruitlet takes the fruitlet address in the slot "RequestServiceComponent" and creates a new instance of this fruitlet in order to perform the request.

Even during run-time the `GenericServer` fruitlet would be able to change the address in the slot "RequestServiceComponent" and thus vary the behavior of the server. This helps to keep the server as open as possible and helps to reuse the fruitlet.

7.2 HTTP Server

This section describes an example of an open, extendable HTTP server (see also the design of another open HTTP server called Jigsaw[1]). We use the `GenericServer` fruitlet of section 7.1 to implement the HTTP server. The server should be open concerning different HTTP protocols and concerning exchangeable parts within the request-resolve-response chain during a task performance.

7.2.1 Design

First of all, our fruitlet has to fit into the "RequestServiceComponent" slot of `GenericServer`, i.e. it has to implement the interface `ServiceRequestIfc` (see appendix A).

We call this fruitlet `HTTPVersionChooser`. The main task of this fruitlet is to recognize the HTTP version of the incoming request. Depending on the result it has to load the corresponding "protocol" fruitlet and passes the open communication channel to this fruitlet. `HTTPVersionChooser` defines two slots. The slot "Version09" specifies the address of the HTTP version 0.9 fruitlet and the slot "Version10" specifies the address of the HTTP version 1.0 fruitlet.

Furthermore, we implement fruitlets for restricted HTTP/0.9 and HTTP/1.0 protocol. Both implement the interface `HTTPTaskIfc` (see appendix A) and thus fit into the `HTTPVersionChooser` fruitlet.

We subdivide the job of an HTTP task into three parts. Firstly, a request part which parses the request. Secondly a resolution task, which takes a request part and generates

[1] Jigsaw: `http://www.w3.org/pub/WWW/Jigsaw/`

a response part as a result. In order to keep the server as open as possible we implement all these parts as fruitlets. Thus a `HTTPTask` fruitlet accesses these parts through corresponding interfaces (see appendix A).

7.2.2 Conclusion: HTTP Server

The HTTP server example shows a few fruitlets, that use the general server fruitlet `GenericServer` (see section 7.1).

This example shows how we can free ourselves from protocol dependent server technology and move in the direction of protocol independent server technology. As a matter of principle it is possible to plug new protocols into an existing server (even during run-time). In some cases (e.g. Network Management) it is essential to attain run-time updatable services.

The server is also flexible (open) with respect to "code choosing". For example, it would be possible that an HTTP request can decide itself (or at least suggest) which code fragment it wants to use for its own resolving-step (e.g. by sending a special header field `ResolveCode: <fruitlet-address-for-resolver>` with the request). Our fruitlet approach enables steps in this direction.

7.3 Open Message Organizer

The various message systems on the Internet lead indisputably (amongst other factors) to the current success of the Internet. There are a lot of examples of standard message systems like electronic mail, news groups and bulletin boards. Often, the user has to deal with two or more message systems at a time and thus use different client applications for each message system.

We want to show an example of an open message organizer that can integrate various message systems into one application. Furthermore, the organizer is open to the addition of new message systems.

Message systems will evolve over time, e.g. there will be new message systems using new protocols. From the users point of view the process of fetching, structuring and managing messages remains the same and it would be a nuisance to install a new client for each message system. Fruitlets offer a solution because the provider of a message system is able to provide both the message and the mobile component (fruitlet) for integration into the message organizer environment.

The organizer can even add new parts during run-time. Thus, the user can seamlessly integrate new message systems from different information providers into the organizer (possibly very specialized message services) and mix them, for example with standard message systems.

7.3.1 Design

We restrict ourselves in the current example to the reception mechanisms of message systems. In order to keep the organizer as open as possible we subdivide the process of receiving messages.

We introduce a chain of software abstractions that an incoming message runs through. At the moment we distinguish between four software abstractions:

- *Retriever*: a retriever is able to fetch messages from the Internet. In order to retrieve a message, it uses a standard or a specialized non-standard protocol. Our message organizer can contain arbitrary types of retrievers.
- *Storage*: a storage is the representation of an interface to a non-volatile storage system. For example, it can map a set of messages to a local file system as well as to a remote file system.
- *Folder*: a folder is a visual representation of a set of stored messages. Normally it provides a mechanism to select and display a particular message.
- *FolderRetrieverConnector*: is a piece of software between a retriever and a folder. It receives a message from its retriever and can decide if it should handle the message to its folder. It can also transform or filter (parts of) a particular message before passing it on.

Each of the described abstractions implement a corresponding interface. We introduce the interface `RetrieverIfc` (implemented by any *retriever*), `FolderIfc` (implemented by any *folder*), `StorageIfc` (implemented by any *storage* system) and `ConnectorIfc` (implemented by any *FolderRetrieverConnector*). For a more detailed description see appendix A. Of course we try to design these abstractions as fruitlets. Thus, all the parts have to fulfill the needs to fit into the fruitlet scheme (e.g. implementing `ComponentIfc`).

Through these interfaces it is possible to combine the various fruitlets to become a flexible message organizer. The constraints are: We can only connect a single *FolderRetriever-Connector* to exactly one retriever and exactly one folder. A storage can have exactly one folder and vice versa.

7.3.2 Omo Run-time Connector/Composer

In order to connect and "compose" different fruitlets of the open message organizer during run-time, we equip the system with a little visual connecting tool. This tool helps to import, connect and parameterize message fruitlets (e.g. some fruitlets may need input from the user) during run-time. This little tool makes it possible to influence and change the whole behavior of the open message organizer during run-time. Figure 7.1 shows a screen shot of the Omo connector/composer.

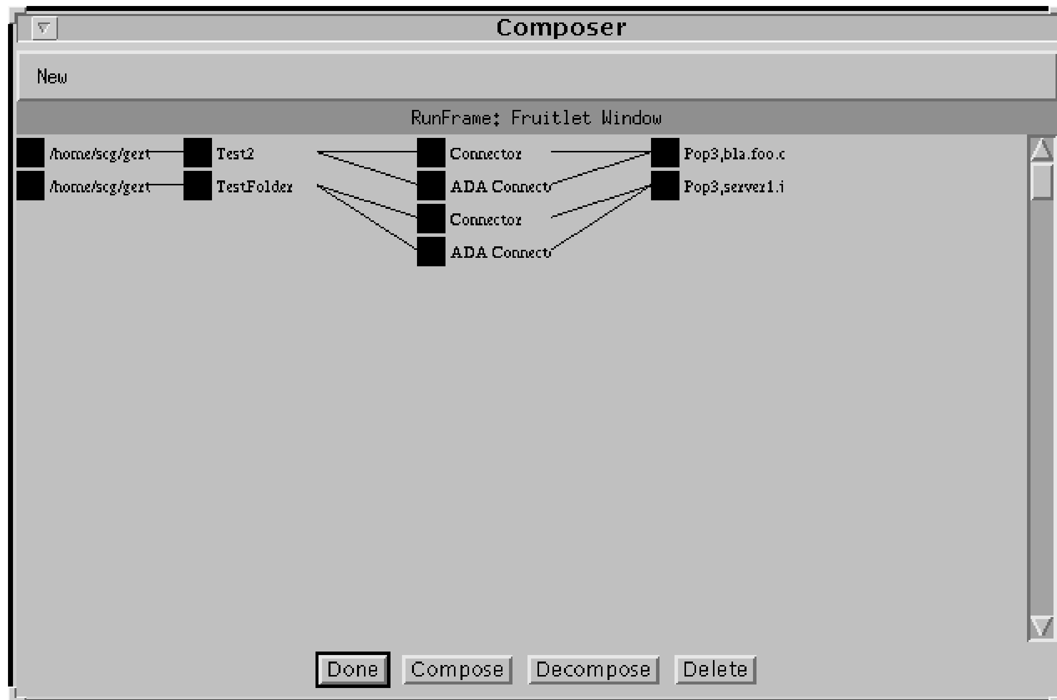

Figure 7.1: Composer/Connector of Omo

7.3.3 The Ada 95 Experiment

Because of the similar properties and goals of Java and Ada 95 (Object-oriented, type safe, concurrent language constructs) it seems to be possible to map the Ada 95 programming language quite well to the virtual machine specification of Java [Taf96, Sun95b, Bar95]. Thus, Intermetrics provides a beta version of an Ada 95 to Java byte-code compiler, called *AppletMagic*.

We use this compiler in order to develop a fruitlet for the open message organizer. We implemented another *FolderRetrieverConnector* fruitlet fully in Ada 95. The functionality is exactly the same as described in section A.3.3.

In order to achieve this goal we had to rewrite a couple of interfaces as Ada 95 source (e.g. `ManagerIfc`, `ComponentIfc`, `OmoIfc`, `ConnectorIfc`, see appendix A). The interface parts to the original Java packages are available as Ada 95 source (e.g. `java.lang`, `java.awt`). Thus it was relatively easy to integrate the *AdaFolderRetrieverConnector* fruitlet into the open message organizer.

This example shows the possibility of decoupled development of fruitlets, even by using another programming language.

7.3.4 Conclusion: Open Message Organizer

The open message organizer shows that our prototype concept fulfills quite a few of the requirements. Firstly, we can show the adaptability of the system with respect to previously unknown message systems. Secondly, we show the heterogeneity of the system with respect to platform independence and also (in a restricted sense) to language independence.

7.4 Conclusion

The examples of this chapter shows some possible applications of our fruitlet approach.

Our approach seems to be general enough to build a wide variety of applications. In contrast to Applets, fruitlets do not expect a graphical user interface and therefore they are more general (see section 7.1).

The HTTP server example also validates that the approach is useful to build open server (and browser) applications for the Internet.

Chapter 8

Conclusion

8.1 Conclusion

The first part of this thesis discussed several requirements of open, flexible software systems. We briefly surveyed different kinds of existing mobile software entities. We restricted ourselves to stateless types of mobile software entities for the rest of this thesis.

In the second part we introduced the notion of a "fruitlet" as a kind of stateless mobile component and we also introduced the term "run-time framework" as a basic software architecture in order to host "fruitlets".

The third part showed a prototype implementation of the run-time framework written in the Java programming language. Furthermore, we designed some examples of open, flexible software applications developed in our "fruitlet" technology.

The following sections summarize the experiences we made with the prototype and the examples concerning the requirements listed in the first part of the thesis.

8.1.1 Adaptability

Our basic software layer (run-time framework) shows a way to improve adaptability and reconfigurability of applications.

For instance, the open message organizer (see section 7.3) shows a flexible design in order to evolve the application in the direction of new requirements (e.g. new message protocols, new message systems). The run-time framework also enables the extension of applications during run-time (reconfigurability).

Our approach uses a relatively simple composition mechanism, thus fruitlets can only be used in a predefined way (using well known interfaces). This fact restrict the (possible) adaptability and evolution of a system.

8.1.2 Heterogeneity

The decision to use the Java technology (especially the Java byte-code), let us improve software systems in the direction of heterogeneity. The introduced examples in chapter 7, show prototypes of software applications which are able to run in heterogeneous networking environments and thus overcome platform and location boundaries. There is some effort going on to improve the performance of the Java byte-code (e.g. using platform dependent "just in time" compilers).

Although there are some problems concerning the Java byte-code (e.g. the byte-code has not exactly the same behavior on different graphical user interfaces), we think it is an excellent approach (although not new) in order to gain heterogeneity within networking environments.

8.1.3 Mobility

Mobility is doubtlessly a basic requirement of modern applications which live in a networking environment. Our approach of "fruitlets" only shows one special aspect of mobility (see section 3.5). "Fruitlets" never transport any state, but carry functionality along a network. In order to attain the full capacity of mobility, we use both stateful mobility (e.g. mobile agents) and stateless mobility. The stateless mobility of fruitlets helps to attain requirements like adaptability and reconfigurability.

Even in the case of very static entities within a networking environment (e.g. databases), the concept of stateless mobile entities is very suitable in order to provide mobile access code to such static entities (see section 3.4.2, see also JoQuer, a mobile database query optimizer[1]).

8.1.4 Security

We fit out our fruitlet technology with a basic security concept. The concept is sufficient to control access to basic resources like network and disk of imported fruitlets. The concept is not able to check the amount of used resources (e.g. disk space, CPU cycles). At the time of writing, the Java byte-code does not offer mechanisms to control "resource deprivation". It is clear that the security of mobile entities in general is itself a large research topic and we did not expect a complete solution concerning the security of fruitlets.

8.1.5 Fruitlet Granularity

One important advantage of fruitlets is the freedom of the granularity of a fruitlet design. On the one hand we can use "conventional" design mechanisms in order to construct a fruitlet and put the fruitlet boundary around many classes. On the other hand, we can also abstract one single class as a fruitlet. This fact helps us to reuse existing designs and to apply promising techniques like design patterns.

[1] JoQuer: `http://wwwis.cs.utwente.nl:8080/skow/phd-pub/`

8.2 Open Problems

In this section we summarize some problems, which are apparent at the time of writing.

8.2.1 Security Concept

In contrast to the current security concept of the run-time framework, we tried also another security approach of fruitlets. The idea was to introduce a security policy which depends on the context in which fruitlets are running. For instance, fruitlet C is running as "inner" fruitlet of fruitlet A and also as "inner" fruitlet of fruitlet B. In this case we could have different policies for fruitlet C.

The Java security manager offers some features to determine the current context of classes during run-time. It is possible to examine the current execution stack (stack of involved classes) during a security check. This fact helps us to find out the current context of a fruitlet and thus to implement the described security approach.

If an action within a fruitlet is caused by an independently running thread (e.g. the AWT main thread which is responsible for user events), we can no longer determine the context of the fruitlet, because "outer" fruitlets are no longer involved.

Thus, it was not possible to implement this approach. The thoughts about security concepts led us anyway to the basic question: what should a security concept of mobile components look like? At the moment we are not able to give a definitive answer.

8.2.2 Performance

Concerning the performance, we have two different problems. Firstly, the Java byte-code is interpreted. In contrast to machine code, this fact leads us to a serious loss of performance (see also "Jigsaw performance evaluation"[2]). "Just-in-time" Java compilers are the current answer to this problem and the estimated speed up is about factor 10. Secondly, our technology consists of loosely coupled mobile abstractions. This fact can lead to time delays mainly caused by the networking environment itself. This problem exists predominantly at startup-time of an application.

8.3 Further Work

8.3.1 Run-time Composition

The current approach of fruitlets shows a way to import and compose mobile abstractions into a system's run-time. What we would like to attain are more powerful composition mechanisms during system's run-time. Fruitlets could carry much more meta information and thus enable more powerful compositional features.

[2] Jigsaw: `http://www.w3.org/pub/WWW/Jigsaw/`

8.3.2 Synchronization during Run-time

Another open question is the synchronization policy between fruitlets (possibly active fruitlets) during run-time. If we are able to connect/compose mobile entities during run-time, what synchronization mechanisms do we have to introduce in general?

8.3.3 New Trend: Java Beans

Recently, Sun Microsystems released a draft specification of a component model called "Java Beans"[3]. The model mainly focuses on mobility and platform independency within networking environments.

A Java Bean is a stateful run-time entity (see section 3.5). Java Beans are collections of connected objects, which are movable. Every Java Bean has an outermost object which serves as a gateway to the encapsulated collection of objects.

A Java Bean has two different types of interfaces. Firstly a design time interface in order to inspect a "bean" during design time and secondly a run-time interface in order to influence the outermost object of a bean during run-time.

At the time of writing, the available information about Java Beans show us some related ideas between fruitlets and Java Beans. In contrast to fruitlets Java Beans are stateful mobile entities, but it seems that the technology also addresses some of our basic ideas of stateless mobility.

A more exhaustive survey of Java Beans could help to uncover the relationship between both approaches.

8.4 Acknowledgments

Many people have contributed to the realization of my thesis and I am very grateful to all of them.

I would like to thank especially Prof. Dr. Oscar Nierstrasz (for supervising my thesis), Karl Guggisberg (for advising me, for many really helpful discussions, for many valuable remarks, and for his patience), and all the other members of the Software Composition Group of University of Berne.

[3] Java Beans: `http://splash.javasoft.com/beans/`

Appendix A

Use Cases: Design

This chapter is the appendix of chapter 7. We give a more detailed overview of the examples introduced in chapter 7.

As a first example, we explain the details of the general server fruitlet (see section 7.1).

In addition we also explain the design of the open HTTP server, which fits into the basic architecture of example one (see section 7.2).

The last part of this appendix introduces the design of the open message system (see section 7.3).

A.1 General Internet Server

The basic architecture of a TCP/IP based server is quite simple and remains the same for most of the common protocols. Our server is listening on a socket with a well known port number in order to accept incoming calls. Once a call has been accepted, the server starts a new task with the communication channel as a parameter.

A.1.1 Design

The fruitlet `GenericServer` included in the package `unibe.internet` reflects the implementation of a generic server architecture. `GenericServer` has two slots (see section 6.2.3). Firstly, a slot with name “Port” and secondly a slot with the name “Request-ServiceComponent”. The “Port” slot specifies the port on which our server should listen and the “RequestServiceComponent” slot specifies the fruitlet, which performs an incoming request. This fruitlet has to implement the interface `ServiceRequestIfc`. The `GenericServer` itself implements `ServerIfc`.

`ServerThread` and `ServiceThread` are not fruitlets, they are just associated classes and belong to the fruitlet `GenericServer`. With the class `ServerThread` the `GenericServer` maintains its own thread. In `ServerThread` we find the main loop. For each incoming

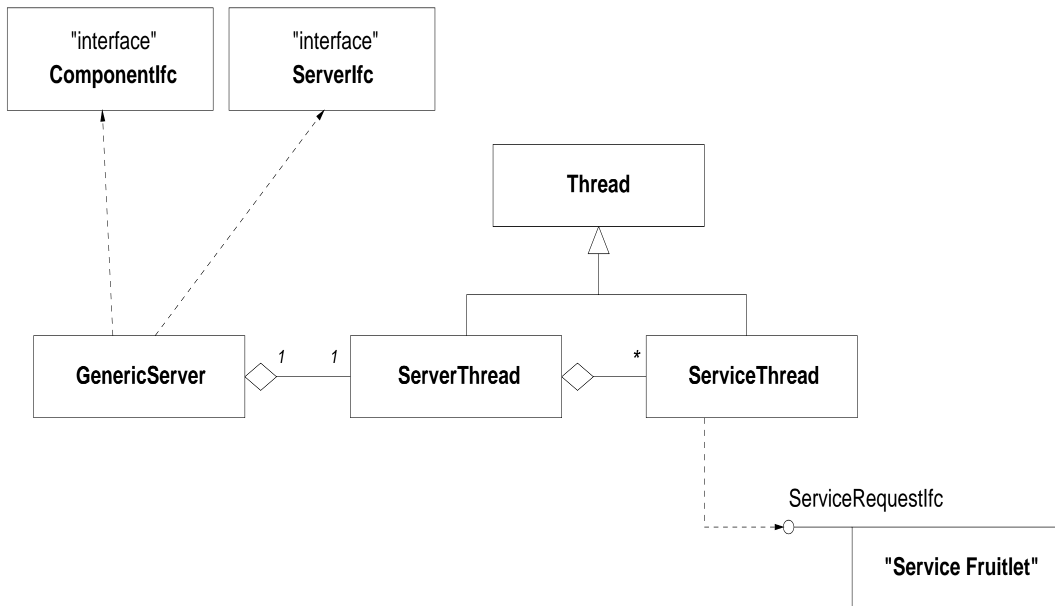

Figure A.1: Class Diagram: Generic Internet Server

call, `ServerThread` starts a new thread `ServiceThread`. `ServiceThread` loads then the "RequestServiceComponent" fruitlet into the server and starts the method `newRequest` (of interface `ServiceRequestIfc`). Thus a new call is separated from the server thread as soon as possible and the server is able to manage new requests.

A.1.2 ServerIfc

This interface contains two methods to start and stop the server. It helps to use the `GenericServer` in other contexts. The `GenericServer` calls `startServer` in its `init` method as a default.

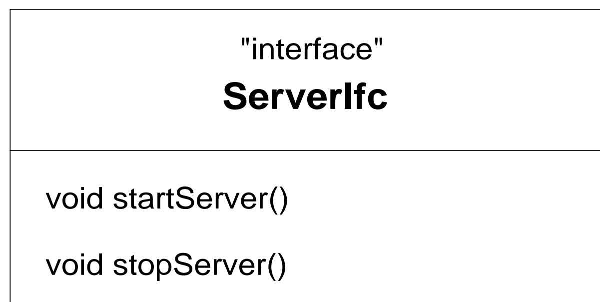

Figure A.2: Interface ServerIfc

```
package unibe.internet.interfaces;

/**
```

```
 * This interface provides methods to start and stop a server.
 */
public interface ServerIfc {

    /**
     * Starts the server.
     */
    void startServer();

    /**
     * Stops the server.
     */
    void stopServer();
}
```

A.1.3 ServiceRequestIfc

`ServiceRequestIfc` is the interface which a request fruitlet must implement in order to fulfill the expected behavior and thus fit into the `GenericServer` fruitlet. The `SocketIfc` is just a basic set of methods to access a socket. In fact we could also use the `java.net.Socket` class instead.

Figure A.3: Interface ServiceRequestIfc

```
package unibe.internet.interfaces;
import  java.io.IOException;

/**
 * This interface is used to send a request to a fruitlet.
 */
public interface ServiceRequestIfc {

    /**
     * Starts a new request.
     *
     * @param socket    the open communication channel
     */
    void newRequest(SocketIfc socket) throws IOException;
}
```

A.2 HTTP Server

This section describes the design of an open HTTP server (see also section 7.2). We use the `GenericServer` fruitlet of section A.1 to implement the HTTP server. The server should be open concerning different HTTP protocols and concerning exchangeable parts within the request-resolve-response chain during a task performance.

A.2.1 Design

First of all, we have to implement a fruitlet which implements the interface `ServiceRequestIfc` (see section A.1.3).

We call this fruitlet `HTTPVersionChooser` (see figure A.4). This fruitlet should be able to recognize the version of the HTTP request. Depending on the result it has to load the corresponding fruitlet and passes the open communication channel to this fruitlet. `HTTPVersionChooser` defines two slots. The slot "Version09" specifies the address of the HTTP version 0.9 fruitlet and the slot "Version10" specifies the address of the HTTP version 1.0 fruitlet.

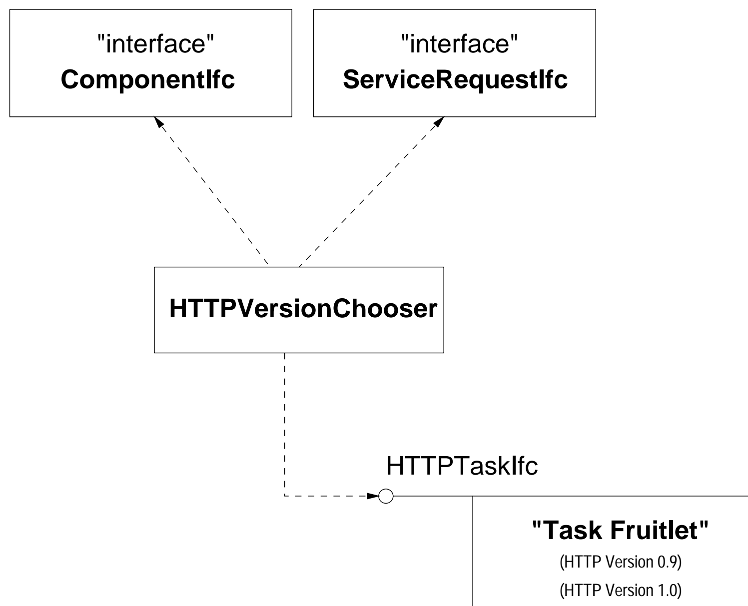

Figure A.4: Class Diagram: HTTPVersionChooser

Furthermore, we implement fruitlets for restricted HTTP/0.9 and HTTP/1.0 protocol. Both implement the interface `HTTPTaskIfc` and thus can be used with the `HTTPVersionChooser` fruitlet.

We subdivide the job of an HTTP task into three parts. Firstly, a request part which parses the request. Secondly a resolution task, which takes a request part and generates a response part as a result. In order to keep the server as open as possible we implement all these parts as fruitlets. Thus a `HTTPTask` fruitlet accesses these parts through

corresponding interfaces.

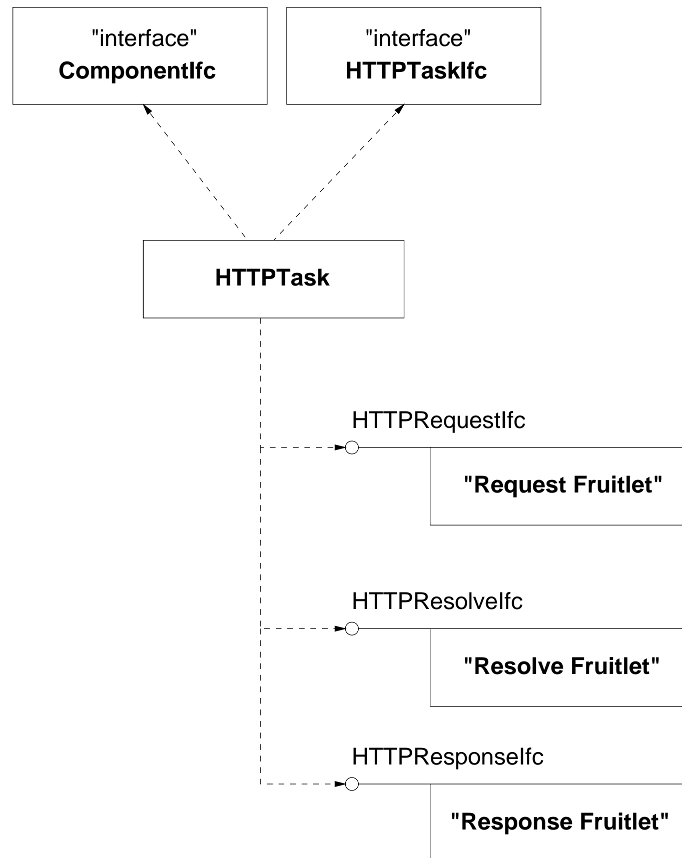

Figure A.5: Class Diagram: HTTPTask (version 0.9/version 1.0)

As an example we show in figure A.6 the class diagram of the request fruitlet, which implements the request part of the HTTP version 1.0.

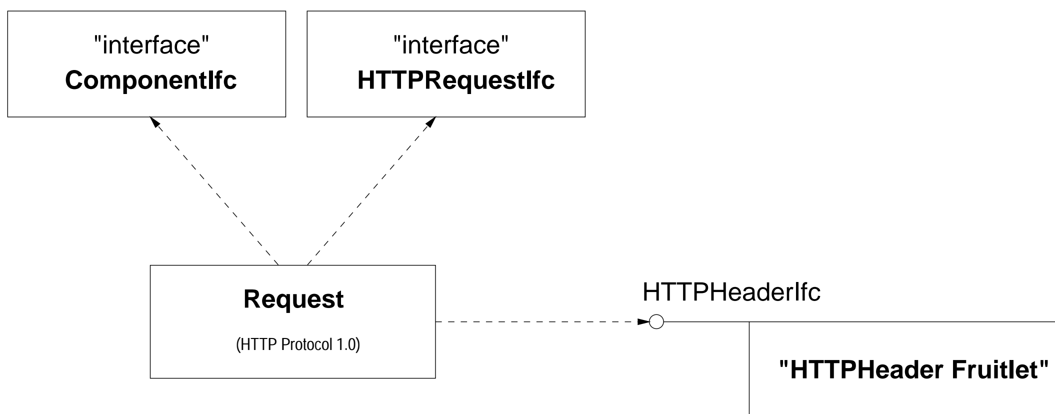

Figure A.6: Class Diagram: HTTPRequest version 1.0

A.2.2 HTTPTaskIfc

The `HTTPVersionChooser` uses a fruitlet with the interface `HTTPTaskIfc`. This interface is quite simple, it just forwards the request and the socket.

Figure A.7: Interface HTTPTaskIfc

```
package unibe.internet.interfaces;
import  java.io.IOException;

/**
 * This interface provides a method to resolve a HTTP task
 */
public interface HTTPTaskIfc {

    /**
     * Resolves a task.
     *
     * @param socket     the open socket
     * @param request    the request string
     */
    void resolveTask(SocketIfc socket, String request);
}
```

A.2.3 HTTPRequestIfc

An `HTTPTask` fruitlet subdivides its jobs into three parts. The first part is to parse the current HTTP request. Depending on the version, this step also includes the reading and parsing of HTTP headers.

```
package unibe.internet.interfaces;
import  java.io.IOException;
import  unibe.internet.http.HTTPDate;

/**
 * This interface provides methods to access a request entity.
 */
public interface HTTPRequestIfc {
```

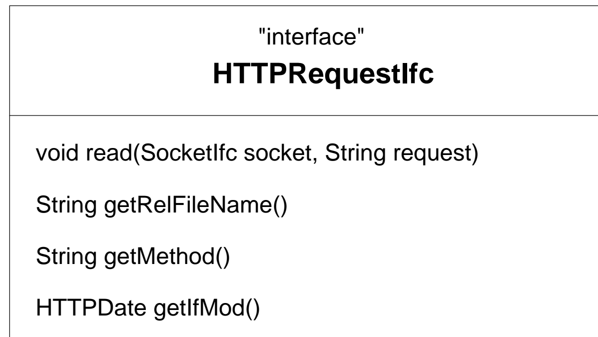

Figure A.8: Interface HTTPRequestIfc

```
    /**
     * Reads the request completely.
     *
     * @param socket     the open socket
     * @param request    the request string
     */
    void     read(SocketIfc socket, String request);

    /**
     * Returns the relative file name of the request.
     */
    String   getRelFileName();

    /**
     * Returns the method of the request.
     */
    String   getMethod();

    /**
     * Returns the date, if a "if-modified-since" header is present.
     */
    HTTPDate getIfMod();
}
```

A.2.4 HTTPResolveIfc

This part of the `HTTPTask` resolves a given request (`HTTPRequestIfc`) and generates a response (`HTTPResponseIfc`).

```
package unibe.internet.interfaces;
import  java.io.IOException;

/**
```

Figure A.9: Interface HTTPResolveIfc

```
 * This interface provides a method in order to resolve a request.
 */
public interface HTTPResolveIfc {

    /**
     * Resolves a request and generates a response.
     *
     * @param req     the request  (input)
     * @param res     the response (output)
     */
    void resolve(HTTPRequestIfc req, HTTPResponseIfc res);
}
```

A.2.5 HTTPResponseIfc

The third part of a `HTTPTask` fruitlet is the response. The response represents a whole HTTP response (eventually including HTTP headers).

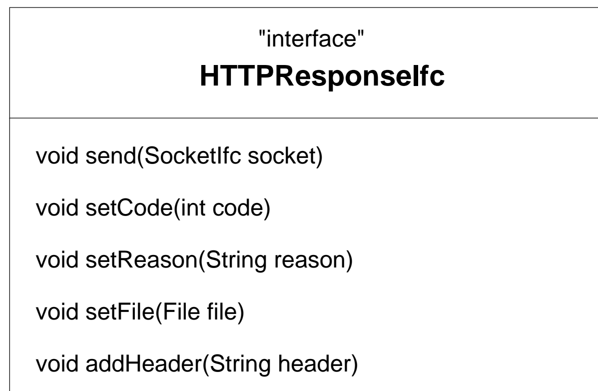

Figure A.10: Interface HTTPResponseIfc

```
package unibe.internet.interfaces;
import  java.io.File;

/**
 * This interface provides methods to access a response entity.
```

```
 */
public interface HTTPResponseIfc {

    /**
     * Sends the whole response to an open socket.
     *
     * @param socket       the open socket
     */
    void  send(SocketIfc socket);

    /**
     * Sets the return code of the response.
     *
     * @param code         the code
     */
    void  setCode(int code);

    /**
     * Sets the reason-phrase of the response.
     *
     * @param reason       the reason-phrase
     */
    void  setReason(String reason);

    /**
     * Sets the file to send.
     *
     * @param file         the file to send
     */
    void  setFile(File file);

    /**
     * Adds a header to the response.
     *
     * @param header       the header
     */
    void  addHeader(String header);
}
```

A.2.6 Conclusion: HTTP Server

The design of our little HTTP server shows that it is possible to build open, extendable servers with our fruitlet approach. It seems that the design is quite flexible concerning the openness and the adaptability of the HTTP server.

A.3 Open Message Organizer

We want to show an example of a system in order to handle different message systems on the Internet (see section 7.3) within the same application.

A user should also be able to integrate new message systems from different information providers into the message system (thus Open Message Organizer).

A.3.1 Design

We restrict ourselves in the current implementation to the reception mechanisms of message systems. We subdivide the process of receiving messages into multiple parts:

We introduce a chain of software abstractions that an incoming message runs through. At the moment we distinguish between four software abstractions:

- *Retriever*: a retriever is able to fetch messages from the Internet. In order to retrieve a message, it uses a standard or a specialized non-standard protocol. Our message organizer can contain arbitrary types of retrievers.
- *Storage*: a storage is the representation of an interface to a non-volatile storage system. For example, it can map a set of messages to a local file system as well as to a remote file system.
- *Folder*: a folder is a visual representation of a set of stored messages. Normally it provides a mechanism to select and display a particular message.
- *FolderRetrieverConnector*: is a piece of software between a retriever and a folder. It receives a message from its retriever and can decide if it should handle the message to its folder. It can also transform or filter (parts of) a particular message before passing it on.

We access each of the described abstractions through a corresponding interface. We introduce the interface `RetrieverIfc` (implemented by any *retriever*), `FolderIfc` (implemented by any *folder*), `StorageIfc` (implemented by any *storage* system) and `ConnectorIfc` (implemented by any *FolderRetrieverConnector*). All these abstractions are designed as fruitlets.

Through these interfaces it is possible to combine the various fruitlets to become a flexible message organizer. The constraints are: We can only connect a single *FolderRetriever-Connector* to exactly one retriever and exactly one folder. A storage can have exactly one folder and vice versa.

Figure A.11 shows an initial approach and overview of the open message organizer.

Because all of the mentioned fruitlets of the open message organizer show the same behavior with respect to some tasks, we decided to introduce a generic interface `MessageSysCompIfc`. The interface `MessageSysCompIfc` collects methods like `delete`,

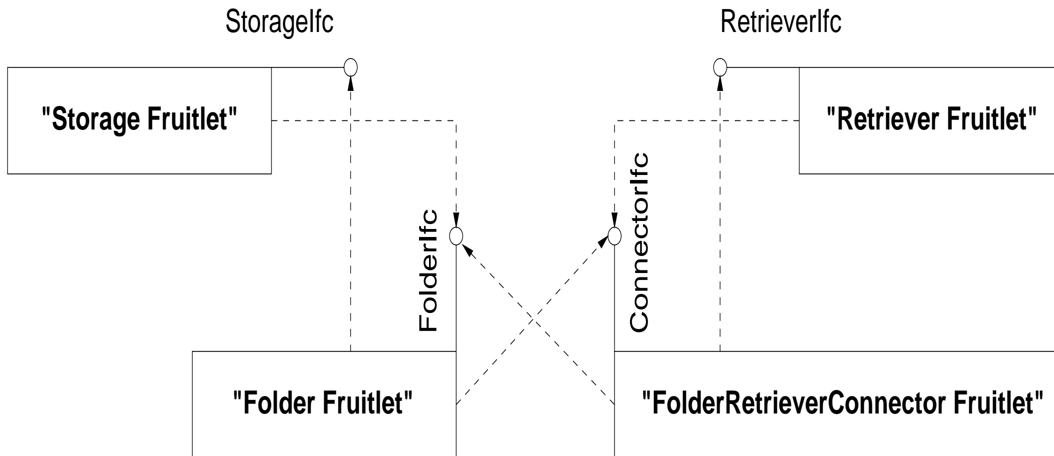

Figure A.11: Class Diagram: Open Message Organizer

`connect` and `disconnect`, which we expect all the message system fruitlets to support. Furthermore, the `MessageSysCompIfc` allows the message system fruitlets to be integrated into the visual connector tool of the open message organizer (see section 7.3.2), by providing access to a graphical component (see method `getView()`). It is also necessary that all the message system fruitlets have access to the open message organizer.

Furthermore, we use an abstraction to encapsulate messages within the open message organizer. A message consists of the message text itself and the header. We introduce interface `HeaderIfc` and interface `MessageIfc` in order to access these abstractions.

This leads us to the class diagram in figure A.12:

A.3.2 Interfaces

We introduce in this section all the interfaces of the open message organizer.

MessageSysCompIfc

```
package unibe.omo.interfaces;
import  unibe.interfaces.ManagerIfc;

/**
 * This interface is the basic access plug of
 * fruitlets which belong to the open message organizer.
 */
public interface MessageSysCompIfc {

    /**
     * Initializes the message system fruitlet.
     *
```

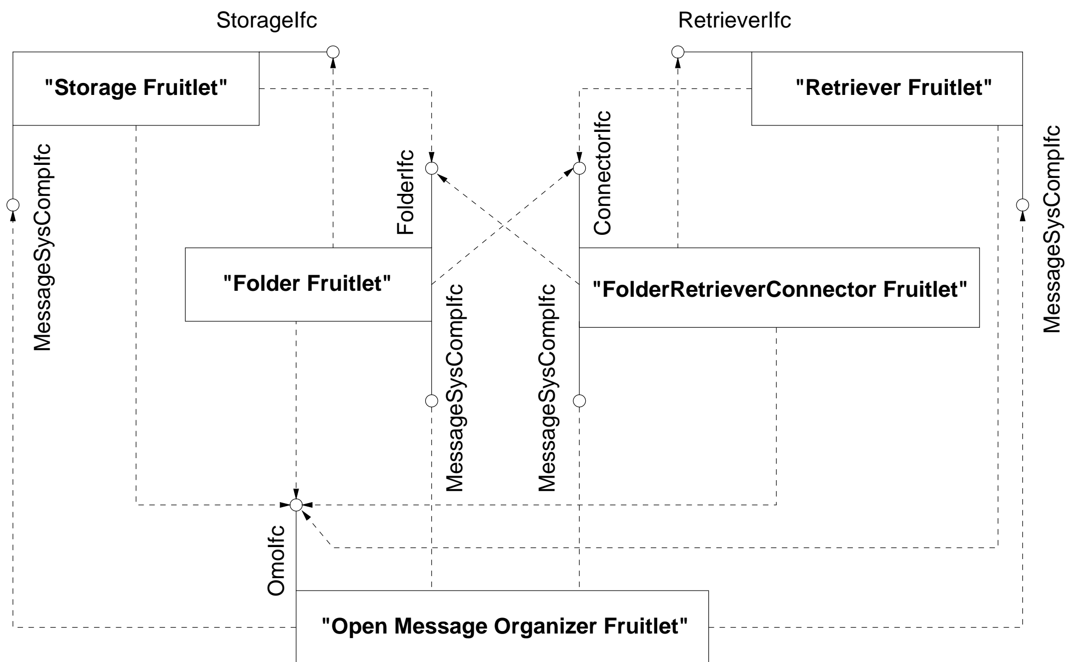

Figure A.12: Class Diagram: Open Message Organizer

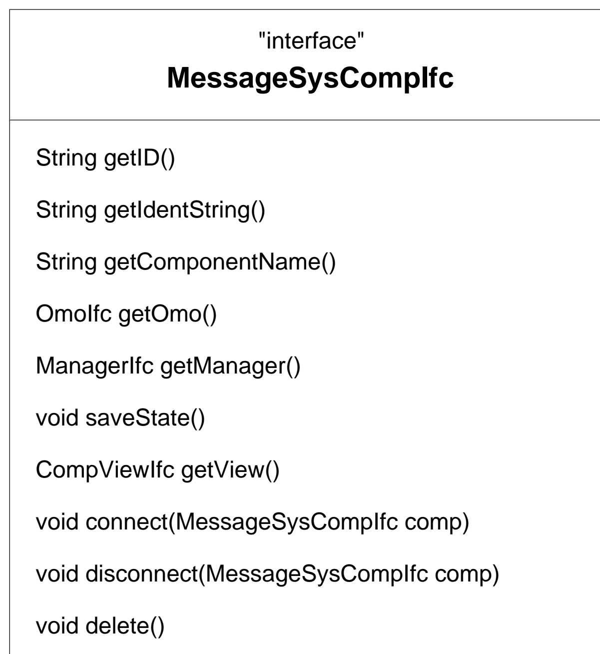

Figure A.13: Interface MessageSysCompIfc

```
 * @param omo            reference to the open message organizer
 * @param componentName  the name of this fruitlet
 * @param id             the ID of this fruitlet
 */
void        initComp(OmoIfc omo,
                     String componentName, String id);

/**
 * Returns the id.
 */
String      getID();

/**
 * Returns an identification String.
 */
String      getIdentString();

/**
 * Returns the fruitlet name.
 */
String      getComponentName();

/**
 * Returns the reference to the open message organizer.
 */
OmoIfc      getOmo();

/**
 * Returns the reference to the system wide manager.
 */
ManagerIfc  getManager();

/**
 * Saves the state of this fruitlet.
 */
void        saveState();

/**
 * Returns the graphical representation of this fruitlet.
 */
CompViewIfc getView();

/**
 * Connects this fruitlet to another fruitlet in
 * the open message organizer.
 * Returns true on success.
```

```
     */
    boolean     connect(MessageSysCompIfc comp);

    /**
     * Disconnects this fruitlet from another fruitlet in
     * the open message organizer.
     * Returns true on success.
     */
    boolean     disconnect(MessageSysCompIfc comp);

    /**
     * Deletes the instance of the fruitlet.
     * Returns true on success.
     */
    boolean     delete();
}
```

RetrieverIfc

Figure A.14: Interface RetrieverIfc

```
package unibe.omo.interfaces;

/**
 * This interface provides the retriever functionality.
 */
public interface RetrieverIfc {

    /**
     * Retrieves new messages. For each new message the retriever
     * calls insertMessage() of the connected FolderRetrieverConnectors
     * (using interface ConnectorIfc).
     */
    void   getNewMessages();
}
```

ConnectorIfc

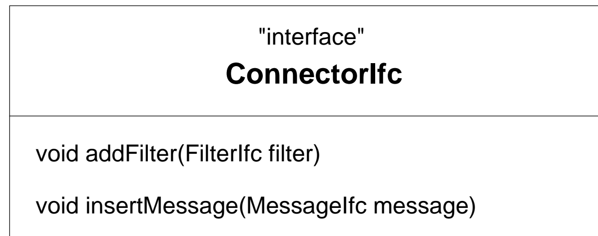

Figure A.15: Interface ConnectorIfc

```
package unibe.omo.interfaces;

/**
 * This interface provides the basic functionality of
 * a FolderRetrieverConnector.
 */
public interface ConnectorIfc {

    /**
     * Adds a filter to the connector.
     */
    void        addFilter(FilterIfc filter);

    /**
     * Passes a message to the connector. If the message passes
     * through all of the connector's filters, the connector calls up
     * insertMessage() from its folder (using interface FolderIfc).
     *
     * @param message   the message
     */
    void        insertMessage(MessageIfc message);
}
```

FolderIfc

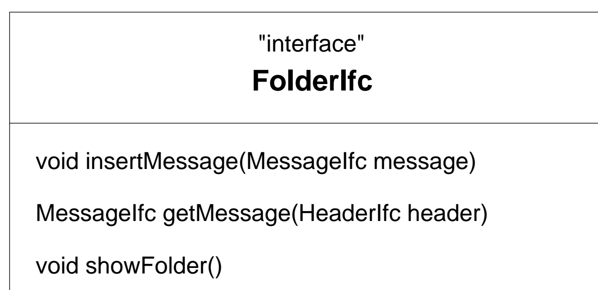

Figure A.16: Interface FolderIfc

```
package unibe.omo.interfaces;

/**
 * This interface provides the basic functionality of
 * a Folder.
 */
public interface FolderIfc {

    /**
     * Inserts a message in the folder.
     *
     * @param message   the message.
     */
    void        insertMessage(MessageIfc message);

    /**
     * Returns a message.
     *
     * @param header    the header.
     */
    MessageIfc  getMessage(HeaderIfc header);

    /**
     * Shows the graphical representation of the folder.
     */
    void        showFolder();
}
```

StorageIfc

Figure A.17: Interface StorageIfc

```
package unibe.omo.interfaces;
import  java.util.Enumeration;
```

```
/**
 * This interface provides the basic functionality of
 * a storage.
 */
public interface StorageIfc {

    /**
     * Returns an enumeration of the headers in this storage.
     */
    Enumeration getHeaders();

    /**
     * Returns a message.
     *
     * @param header    the header of the message.
     */
    MessageIfc  getMessage(HeaderIfc header);

    /**
     * Saves a message in the storage
     *
     * @param message   the message.
     */
    void        saveMessage(MessageIfc message);

    /**
     * Deletes a message.
     *
     * @param header    the header of the message.
     */
    void        deleteMessage(HeaderIfc header);
}
```

OmoIfc

```
package unibe.omo.interfaces;
import  java.lang.String;
import  java.io.DataInputStream;
import  java.io.DataOutputStream;
import  java.util.Hashtable;
import  java.awt.Frame;

/**
 * This interface provides general access to the
 * open message organizer.
 */
```

Figure A.18: Interface OmoIfc

```
public interface OmoIfc {

    /**
     * Returns an input stream, using the id of a message
     * system component.
     *
     * @param id     the ID of a message system component.
     */
    DataInputStream getInputStream(String id);

    /**
     * Returns an output stream, using the id of a message
     * system component.
     *
     * @param id     the ID of a message system component.
     */
    DataOutputStream getOutputStream(String id);

    /**
     * Saves the state of the open message organizer.
     */
    void save();

    /**
     * Starts a message system component.
     *
     * @param name   the name of the component (fruitlet address).
     */
    void startComponent(String name);

    /**
     * Deletes a message system component from the
     * open message organizer.
     *
     * @param comp   the message system component.
     */
    void delete(MessageSysCompIfc comp);

    /**
     * Lists available retrievers
     */
    void listRetriever();

    /**
     * Lists available storages
     */
```

```
void listStorage();

/**
 * Lists available folders
 */
void listFolder();

/**
 * Lists available FolderRetrieverConnectors
 */
void listConnector();

/**
 * Shows the composer/connector
 */
void showComposer();

/**
 * Returns the base directory of the open message organizer
 */
String getBaseDir();

/**
 * Creates a unique id within the open message organizer
 */
String getNewID();

/**
 * Returns the message system component
 *
 * @param id         the id of the message system component
 */
MessageSysCompIfc getRef(String id);

/**
 * Returns the visible folder structure of the
 * open message organizer.
 */
FolderNodeIfc getNodes();

/**
 * Inserts a folder to the visible folder structure of the
 * open message organizer.
 *
 * @param folder     the folder
 */
```

```
    void insertNode(FolderIfc folder);

    /**
     * Connects two message system components together.
     *
     * @param first      the first component
     * @param second     the second component
     */
    void connectComponents(MessageSysCompIfc first,
                           MessageSysCompIfc second);

    /**
     * Disconnects two message system components.
     *
     * @param first      the first component
     * @param second     the second component
     */
    void disconnectComponents(MessageSysCompIfc first,
                              MessageSysCompIfc second);

    /**
     * Adds the visible part of a message system components to
     * the composer/connector of the open message organizer.
     *
     * @param view       the view
     */
    void add(CompViewIfc view);

    /**
     * Returns the graphical frame of the open message organizer.
     */
    Frame getFrame();
}
```

MessageIfc

```
package unibe.omo.interfaces;
import  java.lang.String;

/**
 * This interface provides the basic access to
 * message entities.
 */
public interface MessageIfc {

    /**
```

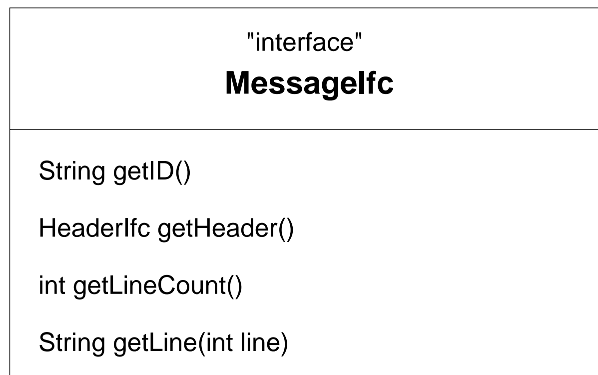

Figure A.19: Interface MessageIfc

```
     * Returns the ID of the message.
     */
    String    getID();

    /**
     * Returns the header of the message.
     */
    HeaderIfc getHeader();

    /**
     * Returns the number of lines of the message.
     */
    int       getLineCount();

    /**
     * Returns a line of the message.
     *
     * @param line     the line number
     */
    String    getLine(int line);
}
```

HeaderIfc

```
package unibe.omo.interfaces;
import  java.lang.String;
import  java.util.Enumeration;

/**
 * This interface provides the basic access to
 * headers of message entities.
 */
```

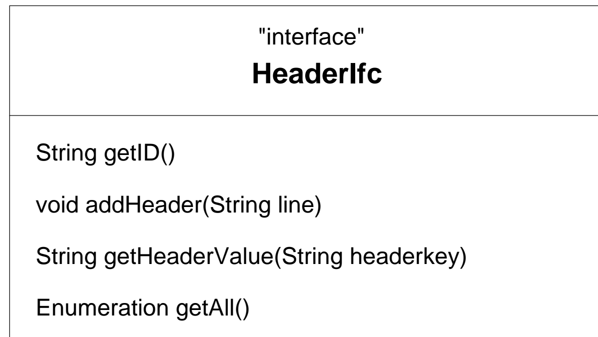

Figure A.20: Interface HeaderIfc

```
public interface HeaderIfc {

    /**
     * Returns the ID of the message.
     */
    String      getID();

    /**
     * Adds a header to the message.
     *
     * @param line          the header line
     */
    void        addHeader(String line);

    /**
     * Returns the value of a header entry.
     *
     * @param headerkey     the key
     */
    String      getHeaderValue(String headerKey);

    /**
     * Returns an enumeration of the headers.
     */
    Enumeration getAll();
}
```

A.3.3 Examples of Omo fruitlets

In order to show the functionality of the open message organizer prototype we implement four fruitlets. With these four fruitlets it is possible to configure and run the prototype.

POP3 Retriever

The Post Office Protocol (POP for short) [MR96] is one of the most popular Internet protocols in order to retrieve electronic mail. We implement an example of a restricted POP3 client. In order to fetch new messages constantly from a POP server, our retriever maintains its own thread. The thread is dormant during a defined time period before calling the `getNewMessages()` method using the `RetrieverIfc`.

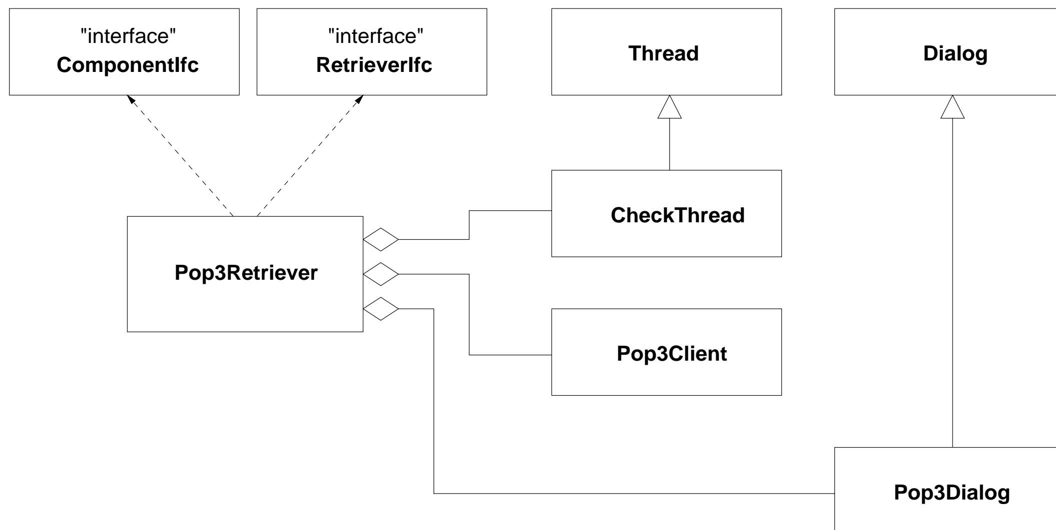

Figure A.21: Class Diagram: POP3 Retriever

Standard FolderRetrieverConnector

The standard *FolderRetrieverConnector* can contain zero or more filters. If an incoming message passes all the filters the standard *FolderRetrieverConnector* inserts the message into its folder.

Standard Folder

We show a simple folder example, which lists the currently available messages. The folder can display the messages in a text box.

Disk Storage

A storage fruitlet of the open message organizer is responsible for managing arrived messages in a non-volatile store. We implement a simple storage fruitlet, which stores the messages on the local file system. We simply use the ID of the storage fruitlet to create a directory. Within this directory we store the messages as text files.

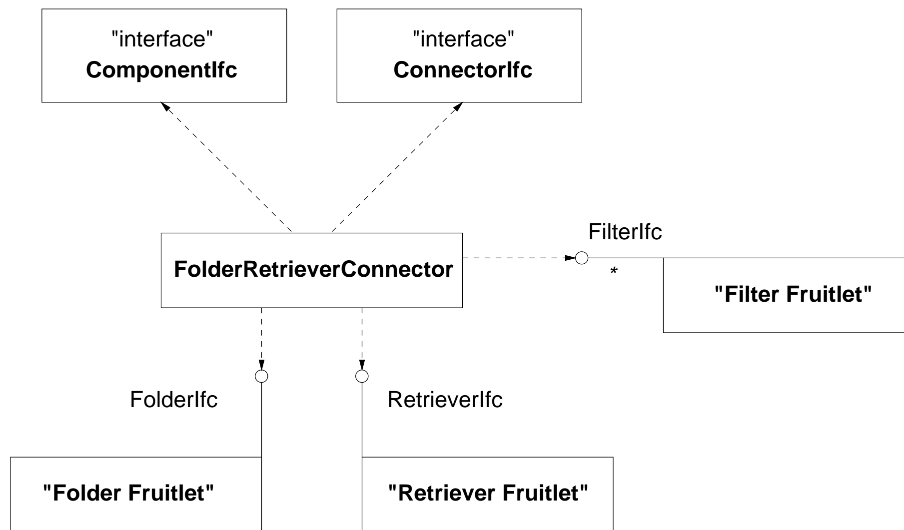

Figure A.22: Class Diagram: FolderRetrieverConnector

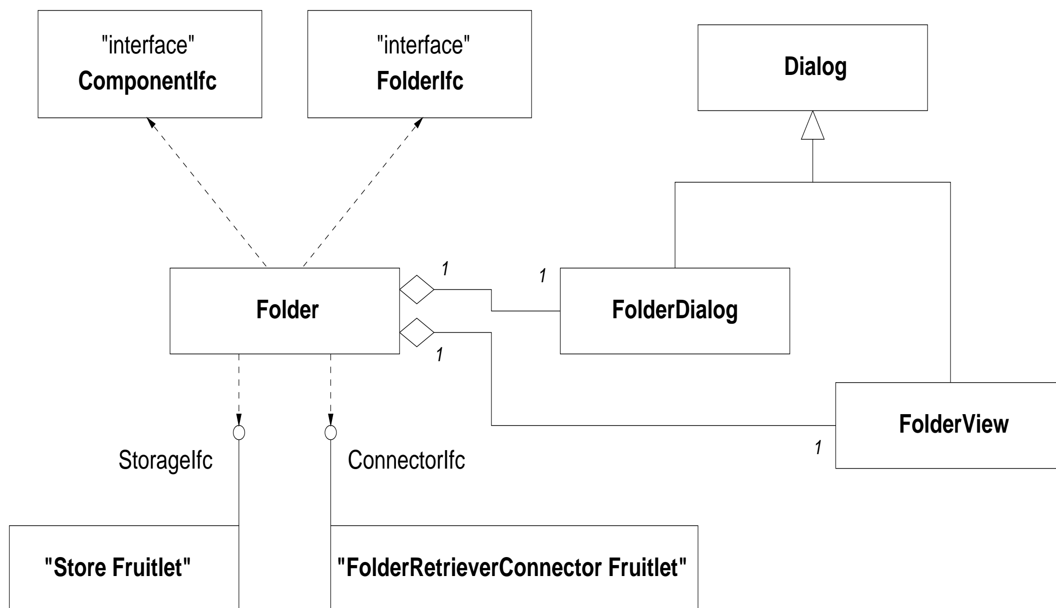

Figure A.23: Class Diagram: Standard Folder

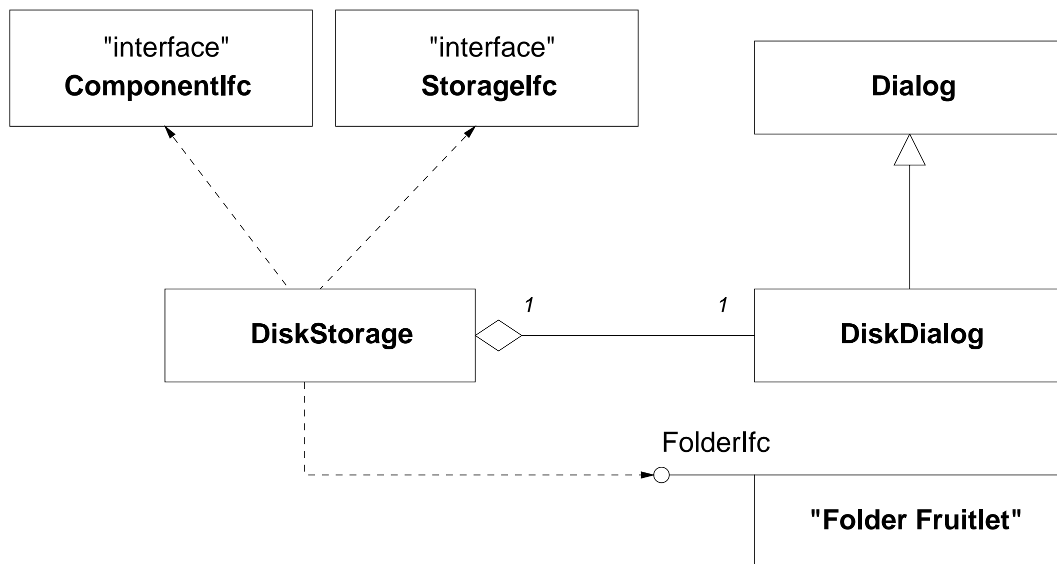

Figure A.24: Class Diagram: Disk Storage

A.3.4 Conclusion: Open Message Organizer

The design of the open message organizer (see also section 7.3) shows that the fruitlet concept provides adaptability concerning the openness of the message organizer.

Appendix B

A Closer Look at Java

This section should give a more detailed overview of Java, for readers who wish to go deeper into the Java topic or who are not very familiar with Java, yet.

This chapter discusses the following topics of Java: the programming language, the object oriented properties of Java, details of Java's class loading mechanism, the virtual machine, the security concept of Java and Java's core class libraries.

B.1 Introduction

The growth of networks superseded stand-alone non-networked computing in the nineteen eighties. The idea of distributed computing replaced the stand-alone and local use of resources and led us to the model of client/server. Client computers were able to use services from other computers, called servers. Obviously this model helped to decrease the cost of the client computers and let us concentrate the high cost services on one machine.

The early nineties saw a incredible growth of wide area networks (WAN), especially the growth of the Internet. These networks allow people to connect easily to other machines around the globe. Between two machines, we may have very different media of physical transport layers, like telephone lines, optical fiber or wireless connections. The response time of such a WAN is hard to estimate and can change during any given period of time dramatically. The client/server model loses its power more and more in such an environment, because special services may no longer be accessible within a set (or useful) period of time.

One approach to the given problems is the idea of mobile code. In such a scenario people try to transfer code from one machine (server) to another (client). The client itself is then able to run the code and thus provides a requested service itself with minimal network access during the computation. The service can remain on the client computer as long the client wants to use it.

In early 1995 Sun Microsystem released such a mobile code technology. The concept

includes a new object-oriented programming language, called Java, and libraries to enhance the core language. After the success of the first Java release, the potential of Java becoming an industrial standard for mobile computing is high.

B.2 Basic Concepts of Java

Java is not only the name of a new object-oriented programming language, but also reflects the whole mobile code paradigm from Sun Microsystem. The basic idea of Java is very old (UCSD p-system[1]). The main problem of any mobile code system is to get a representation of code which is mainly independent of the platform it runs on.

In doing so, Sun has created a platform independent abstract code representation for compiled Java source. This representation is called Java byte-code. The idea is to introduce interpreters (virtual machines) for each platform which can run the Java byte-code. A compiler can create portions of such Java byte-code, and the code is ready to travel through a network and is able to run on different computers within this network. With some restrictions it is also possible to compile other languages than Java into the Java byte-code (e.g. Ada 95 [Taf96]). This fact leads us to the interoperability between different programming languages.

Mobile code concepts raise a lot of questions concerning the safety and security of such a system. We discuss the security solutions of Java in section B.7.

B.3 The Java Programming Language

The object-oriented programming language Java is related to well-known languages like Smalltalk, Objective-C and C++. The syntax of Java is similar to the C++ syntax, but there are some important differences. In order to obtain a safe language, Java does not use any pointers (see section B.7). The Java language provides an integrated module concept in order to get a better structure in the source code.

Java is strongly typed. Thus, every variable and every expression must have a type. Java checks these types by rigorous static type checking.

B.3.1 Java's Object Model

Like other common object-oriented programming languages, Java uses a class based approach. Classes are descriptions for objects, and every object must be instantiated from a class. Classes can contain variables and methods, both can either be class-based or object-based. Every method is virtual and there are no possibilities to overload operators.

Java does not support parameterized types, like the C++ "template" mechanism ("category" in Objective-C). There is some effort going on to introduce parameterized types

[1]UCSD: `http://infopad.eecs.berkley.edu/CIC/archive/cpu.history.html#pMachine`

into the Java core language [BLM96] (see also the Pizza project[2]).

B.3.2 Encapsulation

The Java environment forces each public class to be declared in a single file. This mechanism helps to separate specification from implementation. Java compilers translate Java class sources into byte-code representations of a class. Such a class is stored in a single file and contains a platform independent representation in the Java byte-code format of a class. Java helps to group related classes into packages, where each package defines a new name space for its classes.

B.3.3 Inheritance

Java only provides a single inheritance model for implementation. The notion of interfaces allows a multiple inheritance hierarchy beside the single inheritance hierarchy of the classes. Interfaces can specify a set of (abstract) method signatures (protocols). Every class is able to implement any number of interfaces. Thus, two non-related classes can implement to the same protocol.

Java introduces an new keyword “final” for classes, methods and variables in order to restrict subclassing and overriding respectively. This notion was also introduced because of security reasons.

B.3.4 Polymorphism

The only polymorphic mechanism provided by Java is method overriding. A subclass can re-implement a method of its ancestors classes and thus change the behavior of the subclass by re-using code from its ancestors classes.

B.3.5 Dynamism

Java allows for dynamic loading and binding of classes and instantiations and thus for the dynamic extension of any program during run-time. The byte-code representation of Java classes allows Java to keep encapsulation until run-time. The executable code of a Java application is not a big binary file, but consists of all classes, which belong to the application. The Java run-time loads all the classes into the memory as and when they are needed. The run-time system must check imported classes in various ways to ensure integrity and safety of the run-time system itself and of the environment. Similar, the run-time system is able to fetch classes from the network or from any other source. Because there is no registration of class names at the time of writing, this leads to a name-space problem. The current way to reduce the problem is to encapsulate classes in packages and use the naming convention from Sun (class names should start with `companyname.packagename.`).

[2]Pizza project: `http://www.dcs.gla.ac.uk/~wadler/topics/pizza.html`

B.4 The Java Virtual Machine

The Java virtual machine is the run-time system of Java [Sun95b]. The virtual machine is able to execute programs written in Java byte-code. The Java byte-code itself recognizes about 200 machine instructions. The virtual machine acts as an interpreter for the byte-code instructions. The virtual machine consists of registers, operand stack and an execution environment, the garbage collected heap and exception handling is also maintained on the level of the virtual machine.

The virtual machine is important to obtain some of the security features of Java (see section B.7).

B.5 Concurrency

Java provides language-level concurrency mechanisms. The class *Thread* of the core package *java.lang* is the basic unit of concurrency. A *Thread* is the basic notion of sequential computations. The Java run-time is fully multi-threaded. The *Thread* concept helps to integrate the concurrency concept seamlessly into the language.

B.6 Class Loading Mechanism

One important and interesting part of Java is the class loading mechanism. As we discussed in one of the previous sections, the Java run-time can load new classes into the run-time system. The run-time uses so called class loaders to import new classes. For every class, the run-time knows which loader is responsible for this class. Classes from the local file system can be imported with the default class loader.

Normally, classes depend on one or more other classes or interfaces and this implies the knowledge of the related classes/interfaces to instantiate an object of a particular class. Let's assume the following example: we want to load class S, which is a subclass of B. S also has a private variable of type class C. If we try to instantiate an object of class S, we also need to have the knowledge about class B and class C. The Java run-time uses the following strategy to solve this problem: if the class S requires other classes (say B), the run-time tries to get the class definition of B in the same manner as it received the definition of class S. This means the run-time uses exactly the same class loader as with class S but asks it for a definition of B.

Every standalone Java application is able to introduce new class loaders. A class loader can implement the way a class is retrieved from any source (e.g. using HTTP protocol via Internet to retrieve a class). And the class loader can also implement the strategy to obtain related classes. One strategy for example is to first ask the local file system for a class definition, and if this fails, ask a host on the network for the definition.

B.7 Safety and Security

Java's mechanism to extend the type system during run-time, makes it necessary to introduce safety and security mechanisms, firstly in the language and secondly in the run-time system as well.

The Java system provides several layers of safety and security for a stand alone application. An overview of the layers can be given as follows (see also section 3.5.4):

- Layer one: the language and the compiler
- Layer two: byte-code verifier
- Layer three: class loader
- Layer four: Security Manager

B.7.1 Language and Compiler

The Java programming language itself provides a first layer of safety. One advantage of Java over other existing programming languages is that Java was designed right from its early days to be a safe language. In Java the size of all primitive data types is well defined. Thus two different, but correct Java compilers will never give different results for program execution. Other languages (e.g. C++) have different definitions of their primitive types, which can be machine and/or compiler dependent. Dangerous pointer arithmetic was strictly removed from the language, thus Java does not support any pointers. In fact, references to objects cannot be modified in a dangerous way. Every cast operation is rigorously checked during compile-time and run-time. For instance, it is not possible to cast an object into an array of bytes and thus it is not possible to access private data areas of an object.

As we mentioned earlier, the Java language is strongly typed. The compiler checks all the references to methods and variables to ensure the correct types. The compiler also rejects all accesses to uninitialized variables of any program.

B.7.2 The Byte-Code Verifier

If we use a trusted compiler, we can ensure all the safety features described in the previous section. Because Java is able to import new types (classes) in the form of Java byte-code from other sources (e.g. networks) we do not know which compiler has produced these code fragments. It is even possible that classes were compiled from other programming languages into Java byte-code. Thus, the Java run-time system performs a series of tests before running any piece of code.

These tests will be performed directly after the loading of a class. Once a class has passed all the tests it can run without restrictions on the interpreter. Thus the byte-code verifier can help to improve the performance of the interpreter because run-time tests for every

instruction can be eliminated. In short, the byte-code verifier passes the code through four verification phases.

- Pass one: check class file format
- Pass two: test *final*, *superclass* constraints
- Pass three: analyzes each method; does data flow analysis
- Pass four: performs some postponed checks of pass three

Pass one occurs when the interpreter loads a class file. It checks some format properties of class files, like the magic number, attribute length and the constant pool. Pass two checks some constraints of the class hierarchy, for example: every class (except *Object*) must have a superclass, *final* classes are not subclassed. Pass three is the most complex test. This pass checks the code of each method and does a data-flow analysis. Pass three ensures that at any given point of the code, no matter what code path is taken to reach that point, some type constraints are valid. For instance: The stack is always the same size and contains the same type of objects, methods are called with appropriate arguments, fields are modified with values of the appropriate type. Pass four does some postponed tests of pass three. For more detailed information about the byte-code verifier and low level security in Java see [Yel96].

B.7.3 The Class Loader

Class loaders of a stand alone application can help to implement security policies. Class loaders can be used to ensure that a unique name space for classes from different sources (e.g. different network hosts) exists. Classes loaded from the local file system (often called built-ins) are located in a separate name space in any case. Thus it is not possible that classes from different name spaces can access each other without control of the system. If a class is requested from a class loader, it should always look in the built-ins for the class first. There is no way to spoof a built-in class by an imported untrusted class.

Any standalone application is allowed to implement its own class loaders at will. The Java system is not longer directly involved. Thus, an application can influence its own security policy by writing an appropriate class loader.

B.7.4 Security Manager

This layer is also housed on the application level. A stand alone Java application is able to fully implement security policies by introducing a *SecurityManager*. A Java application has zero or one security managers. All dangerous operations of the built-ins call methods of the security manager. For instance, the constructor of the class *InputStream* always calls the security manager's method *checkRead(String name)*. Every time, a class tries to open a file via the *InputStream* class, the security manager is called. The security manager offers some methods of evaluating the current execution environment running on

the interpreter and the security manager can decide to allow or disallow a special access request.

The current security manager is called from the following events: read/write the local file system, read/write the network, execute other programs, exit the current application, instantiate a new class loader, read/write system properties, create a top level windows and access threads.

B.8 The Java Libraries

Java provides an initial class hierarchy to the user. The main goal of this class hierarchy is to provide the basic tools to write platform independent software. The classes are grouped into Java packages. Most of the class methods access system functions by using platform dependent native calls. The rest of this section gives a short overview of the Java core packages.

B.8.1 java.lang

The *java.lang* package consist of classes related to the Java core. Classes like *Object*, *Class*, *ClassLoader* and abstractions of the basic data types like *String*, *Integer* and *Character* are placed in this core package.

B.8.2 java.net

Java was born into a networking world, thus one important package of classes is *java.net*. *java.net* includes abstractions of the Berkeley sockets for TCP/IP [Com95] and some basic abstractions for Web-URL's.

B.8.3 java.awt

This is the biggest package of the core library. It reflects a whole abstraction to a windowing toolkit. The awt (abstract windowing toolkit) framework provides a fully platform independent set of classes to build applications using a graphical user interface.

B.8.4 java.io

The *java.io* package implements a set of basic stream classes to access the local file system. *java.io* includes classes like *InputStream*, *OutputStream* and also higher abstractions of streams (e.g. *FileInputStream*).

B.8.5 java.util

This is a collection of useful classes outside of the core package *java.lang*. *java.util* provides abstractions like *Hashtable*, *Vector*, *Enumeration* and *Date*.

B.8.6 java.applet

The main class of this package is *Applet*. An applet is a mini application written in Java, which is able to run as a subprocess in special Web browsers like Netscape 2.0 or HotJava. For a more detailed discussion of applets and Java enabled browsers see section 3.4.

Appendix C

Diagram Notation

We use the "Unified Modeling Language" [BR96] in order to draw all the class diagrams within this thesis. This appendix summarizes the notational shortcuts we mainly used to show our class diagrams. Because we mainly used class diagrams in the thesis, we can restrict ourselves here to the notations of class diagrams. This appendix is not exhaustive and we suppress a lot of details in order to focus on the features we used in the diagrams.

C.1 Introduction

The Unified Modeling Language is a design method for specifying, visualizing, and documenting the artifacts of an object-oriented system under development. The Unified Modeling Language originated of two leading object-oriented methods (Booch method and OMT).

C.2 Classes

Classes are drawn as a solid line rectangle with three compartments. The top compartment contains the name of the class, the middle compartment the attributes, and the bottom compartment a list of operations (see figure C.1).

The attribute and operation compartments can be suppressed to reduce complexity in an overview. Omitting a compartment makes no statement about the absence of operations and attributes. In contrast, an empty compartment explicitly declares that there are no elements in that part.

C.3 Association

In class diagrams associations represent structural relationships between different classes. Most associations are binary, shown as solid lines between classes. Associations may have

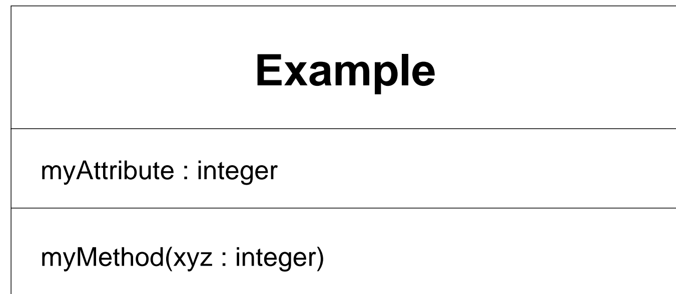

Figure C.1: Class Diagram: Example of a class

names and may have direction arrows. An association could have different names in each direction. Each end of an association is called a role. A role may have a name, the name shows how its class is viewed by the other class. Each role also indicates the multiplicity of its class. The multiplicity shows how many instances of the class can be associated with one instance of the other class (see figure C.2).

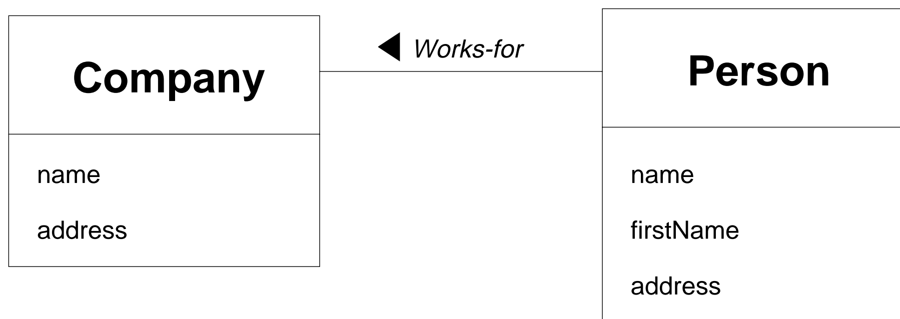

Figure C.2: Class Diagram: Association

Aggregation is a "whole-part" relationship, i.e. that the lifetime of the parts depend on the lifetime of the whole. This relationship is indicated by placing a diamond on the role attached to the whole class (see figure C.3).

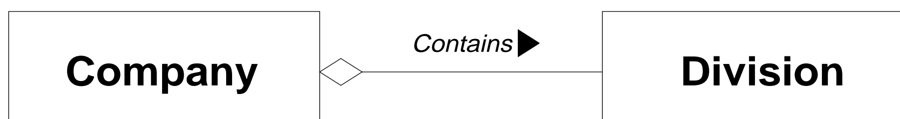

Figure C.3: Class Diagram: Aggregation

C.4 Inheritance

The relationship between a superclass and its subclasses is called *inheritance*. Sometimes this relationship is called *specialization* or *generalization* (depending on whether one is viewing from the superclass to the subclass or vice versa). Inheritance is drawn as a

solid line from the subclass to its superclass with an unfilled triangular arrowhead on the superclass end (see figure C.4).

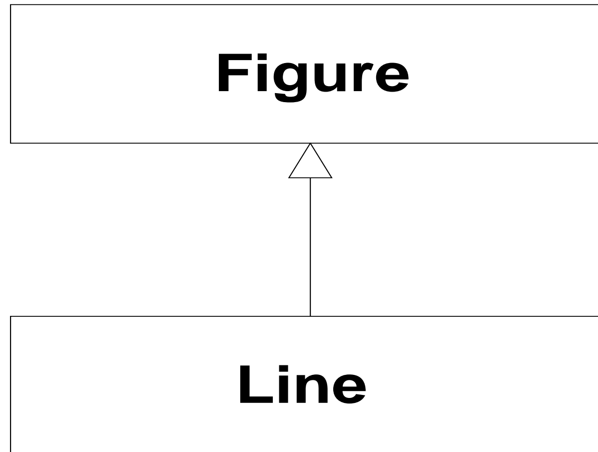

Figure C.4: Class Diagram: Inheritance

C.5 Interfaces

Usually we call a set of function signatures an *interface*. An interface is an articulation of expected behavior and responsibilities of a class. The Unified Modeling Language also provides a notation for interfaces. Because we use Java as implementation language, we use this notation quite often. There exist two notations:

C.5.1 Reified Interface Notation

This notation displays the interface explicitly, drawn as a class but with the stereotype "interface" above the interface name. The fact that a class implements an interface can be shown with a dashed arrow from the class to the interface (see figure C.5). The association may be decorated with the stereotype "conforms".

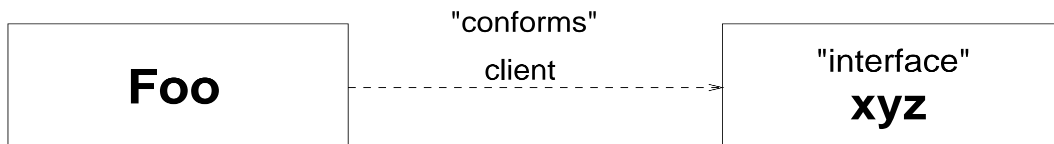

Figure C.5: Class Diagram: Reified Interface Notation

C.5.2 Symbolic Interface Notation

This notation is used to express that a client class is accessing another class using an interface. Figure C.6 shows a client class "Foo" which connects to class "Bar" using the

interface “xyz”.

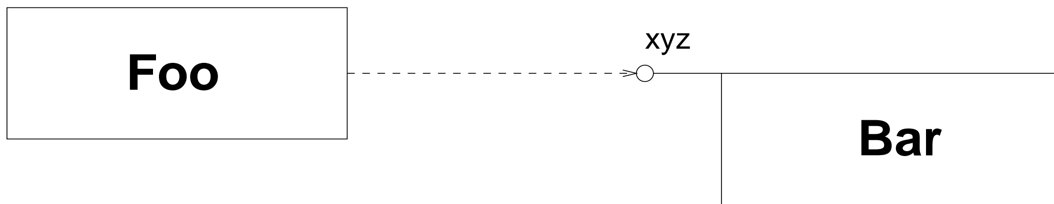

Figure C.6: Class Diagram: Symbolic Interface Notation

Bibliography

[AJDS96] M.P. Atkinson, M.J. Jordan, L. Daynes, and S. Spence. Design Issues for Persistent Java: a type-safe, object-oriented, orthogonally persistent system. Technical report, University of Glasgow, February 1996.

[Bar95] J. Barnes. *Programming in Ada 95.* Addison-Wesley, 1995.

[BLM96] J. A. Bank, B. Liskov, and A. Myers. Parametrized types and java. (submitted for OOPSLA'96), 1996.

[Bor92] N. S. Borenstein. Computational mail as network infrastructure for computer-supported cooperative work. In *CSCW 92 Proceedings*, pages 67–73, 1992.

[BR93] N. Borenstein and M. T. Rose. Mime extensions for mail-enabled applications: application/safe-tcl and multipart/enabled-mail. electronically, November 1993. This document is part of the Safe Tcl distribution.

[BR96] G. Booch and J. Rumbaugh. Unified Modeling Language, Version 0.9. Technical report, Rational Software Corporation, 1996.

[Car95] L. Cardelli. A language with distributed scope. Technical report, Digital Equipment Corporation, Systems Research Center, 130 Lytton Ave, Palo Alto, CA 94301, USA, 1995.

[Com95] D. Comer. *Internetworking with TCP/IP, Vol I, Principles, Protocols and Architecture.* Prentice Hall, 1995.

[CS93] D. Comer and D. Stevens. *Internetworking with TCP/IP Vol III, Client/Server Programming and Application.* Prentice Hall, 1993.

[DFW96] D. Dean, E. Felten, and D. Wallach. Java Security: From Hotjava to Netscape and Beyond. Technical report, Departement of Computer Science, Princeton University, 1996.

[Gen95] General Magic, Inc., 420 North Mary Avenue, Sunnyvale, CA 94086. *Telescript Language Reference*, October 1995.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and R. Vlissides. *Design Patterns.* Addison Wesley, 1995.

[GM95] J. Gosling and H. McGilton. The Java Language Environment, A White Paper. Technical report, Sun Microsytems Computer Corporation, 1995.

[Har95] C. Harrison. Smart Network and Intelligent Agents. 1995.

[HCK95] C. G. Harrison, D. M. Chess, and A. Kershenbaum. Mobile agents: Are they a good idea ? Technical report, IBM Research Division, IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY 10598, March 1995.

[Hoe93] U. Hoelzle. Integrating independently-developed components in object-oriented languages. In O. Nierstrasz, editor, *Proceedings ECOOP'93*, LNCS 707, pages 36–56, July 1993.

[Hoh95] F. Hohl. Konzeption eines einfachen Agentensystems und Implementation eines Prototyps. Master's thesis, University of Stuttgart, August 1995.

[Jol96] V. Joloboff. Java Mobile Code: A White Paper. Technical report, OSF Research Institute, 1996.

[KL96] Bharat K. and Cardelli L. Migratory Applications. Technical report, digital, February 1996.

[Mic95] Microsoft Corporation. *The Component Object Model Specification*, March 1995.

[MN96] T. Meijler and O. Nierstrasz. Beyond Objects: Components. 1996.

[MR96] J. Myers and M. Rose. Post Office Protocol - Version 3. RFC 1939, 1996.

[ND95] O. Nierstrasz and L. Dami. Component-oriented software technology. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, chapter 1, pages 3–28. Prentice Hall, 1995.

[NT95] O. Nierstrasz and D. Tsichritzis, editors. *Object-Oriented Software Composition.* Prentice Hall, 1995.

[Obj92] Object Management Group. *The Common Object Request Broker: Architectures and Specification*, 1.1 edition, 1992.

[Sun95a] Sun Microsystem Computer Corporation. *The Java Language Specification*, 1995.

[Sun95b] Sun Microsystem Computer Corporation. *The Java Virtual Machine Specification*, August 1995.

[Taf96] T. Taft. Programming the Internet in Ada 95. submitted to Ada Europe 96, March 1996.

[van95] G. van Rossum. *Python Tutorial.* Dept. AA, CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands, 1.2 edition, 1995.

[Vin93] S. Vinoski. Distributed Object Computing with CORBA. *C++ Report*, July 1993.

[Weg87] P. Wegner. Dimensions of object-based language design. *ACM SIGPLAN Notices*, 22(12):168–182, 1987.

[WJK96a] M. Weiss, A. Johnson, and J. Kiniry. Distributed Computing: Java, CORBA, and DCE. Technical report, OSF Research Institute, 1996.

[WJK96b] M. Weiss, A. Johnson, and J. Kiniry. Security Features of Java and Hotjava. Technical report, OSF, Research Institute, 1996.

[Yel96] F. Yellin. Low Level Security in Java. WWW, Sun Microsystems, 1996.