INSTITUT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

**Diplomarbeit**

# Dynamic Adaptation of Mobile Code in Heterogenous Environments

Raimund Brandt

INSTITUT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

**Diplomarbeit**

# Dynamic Adaptation of Mobile Code in Heterogenous Environments

Bearbeiter: Raimund Brandt
Aufgabensteller: Prof. Dr. Heinz-Gerd Hegering
Betreuer: Dr. Christian Hörtnagl (am IBM Zurich Reserach Laboratory),
Dipl.-Inf. Helmut Reiser (an der Ludwig-Maximilians Universität)
Abgabetermin: 15. Februar 2001

Ich versichere, dass ich diese Diplomarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15. Februar 2001

.........................................
*(Unterschrift des Kandidaten)*

**The practical work was carried out**
**at IBM Zurich Research Laboratory**

**Abstract**

This work explores the adaptation of mobile code during runtime. There are various aspects to adaptation. Especially mobile agents are faced with changing environments because of traveling in heterogeneous networks. The focus of this work are dynamic re-configuration, context awareness and the architecture of a repository serving exchangeable parts to the mobile code. Results from the implementation of a prototype show that dynamic adaptation can reduce programming effort for mobile code in heterogeneous environments. Furthermore it makes mobile code more efficient in terms of bandwidth and enhances its scalability.

# Contents

# List of Figures

# List of Tables

# Acknowledgements

I would like to thank Prof. Dr. Heinz-Gerd Hegering (TU München) and Dr. Metin Feridun (IBM Zurich Research Laboratory) for the opportunity to carry out the practical part of my thesis at IBM Zurich Research Laboratory.

Special thanks to Dr. Christian Hörtnagl for his excellent help and support accorded to me throughout my thesis. Furthermore, I thank Dipl.-Inform. Helmut Reiser for the constructive guidance he provided me.

Finally I wish to say thank you to all members of the *Distributed Systems and Network Management Group* for the pleasent and motivating atmosphere during my time at IBM.

# Chapter 1

# Introduction

Today's computer networks have complex and heterogeneous structures due to their drastically increased size.

Mobile code seems to be a promising technology which utilizes the advantages and overcomes some disadvantages of such an infrastructure. The motivation for mobile code is to move code to data sources instead of the vice verse.

[LO98] gives a short overview of advantages offered by a special form of mobile code, *mobile agents*. Mobile agents can reduce network traffic because the communication between program and data is done through local interfaces. The local communication also overcomes communication latency, which can enhance performance especially if a big amount of data is exchanged. Another merit of mobile agents is the flexibility to update or to modify protocols. If a protocol layer is only updated or modified on the local host, but not on the remote host, a mobile agent can be sent to the remote host as mediator between the incompatible parties.

Mobile agents enable an application to execute code autonomously. After moving code over the network and initiation of the computation, the connection to the source application can be released and the mobile code continues to execute its task.

This property can make a distributed software system more robust and fault-tolerant, because the potential negative effect of poor links with low bandwidth and low reliability is reduced.

## 1.1 Mobile Code

Mobile code appears in many shapes: as Postscript documents executed on the printer [Ado85], Java applets used for animated and interactive web pages [Fla97] and complex mobile agents executing important network management tasks [Ple99].

The purpose of this section is to define the term *mobile code* and to give an overview of technologies and concepts used in the area of mobile code. Definition of terms and differentiations in this section are based on [FPV98].

In traditional distributed systems, i.e. systems which do not rely on code mobility, the executable parts reside in a computational environment during their entire lifetimes. In distributed systems following the mobile code paradigm, the executable parts can be moved between computational environments.

The main aspects of mobile code are: mobility mechanisms, design paradigms and application domains.

The mobility mechanism enables an application to move executable code between computational environments. The design paradigm benefits from the mobility mechanisms and describe concepts of distributed applications which rely on mobile code. Where mobile code is useful and improves distributed application performance is covered by the section on application domains.

### 1.1.1 Mobility Mechanisms

This section introduces various technologies which enable an application to exploit mobile code. Executable parts as they are used in traditional distributed applications include the executing unit, the sequential flow of computation and the data space, which contains references to resources managed by the local computational environment. The mobility of the executing unit is dealed within the first part of this section. Handling data space is addressed in the second part.

#### Mobility of Code and Execution State

An executing unit consists of the code and the state of the current computation. The executing state comprises private data and control information, e.g. stack call, instruction pointer.

If both code and execution state are moved to a new computational environment, the mobility mechanism is called *strong mobility*. *Weak mobility* represents the transfer of code with some initialization data without recreation of the execution state.

*Strong mobility* is either realized by *migration* or by *remote cloning*. For *migration* the executing unit is suspended and after the transfer of code to the new computational environment, the executing unit is resumed. *Remote cloning* differs from *migration*, because it does not detach the original executing unit from its current computational environment.

*Weak mobility* differes in the direction in which the code moves, the nature of the code, the synchronization and the point of time when the code is actually executed. The direction can either be to the executing unit, *fetch*, or from the executing unit to a computational environment, *ship*. The code is either *stand-alone* or only a *fragment*. Whether the *weak mobility* is *synchronous* or *asynchronous* depends on whether the executing unit moving the code, blocks until the moved code is executed or not. After the move to a new computational environment the code is executed *immediately* or *deferred*.

#### Data Space Management

The executing code references resources managed by the local computational environment. If the executing code is moved, the bindings to the resources must be modified in a such way that the moved code holds valid references in the new computational environment. The resources can be split into *transferable* and *non transferable* resources. For instance, a byte string is *transferable*, but a resource of the type printer is *not transferable* by the mobility mechanism. *Transferable* resources are furthermore either *free* or *fixed*. Though a huge file is transferable it is not desirable by the application because of performance reasons.

Apart from the nature of resources it is also important how the resources are bound to the executing unit. Fugetta et al. [FPV98] describe three kinds of bindings: *by identifier*, *by value* and *by type*.

The strongest binding is the binding *by identifier*. The executing unit references *by identifier* an uniquely identified resource. This binding type is used for resources which cannot be substituted by other equivalent resources. If the executing unit moves to a new computational environment the resource, can be *moved* with the executing unit, presumed that the resource is *transferable* and *free*. If the resource can not be re-allocated in the new computational environment, a *network reference* mechanism must be used. The resource is not moved along with the executing unit. The executing unit holds a reference which points to the resource in the old computational environment. *Network references* imply that the data space is distributed over the network and complicate the management of state consistency.

If other executing units reference a resource and the *by move* mechanism is exploited, *network references* must be assigned to the resting executing units. The *removal* of bindings of resting executing units which reference moved resources leads to undesirable exceptions if the resting executing unit access the void reference.

If a resource is bound *by value* to the executing environment, the most convenient mechanism of data space management is *by copy*. The value of the resource is copied to the computational environment. An alternative is the data space management *by move*, as described for bindings by identifier. It may also be applied, but especially if a resource is referenced by more than one executing units, network references must be assigned to the remaining executing units with the drawback of the distributed data space.

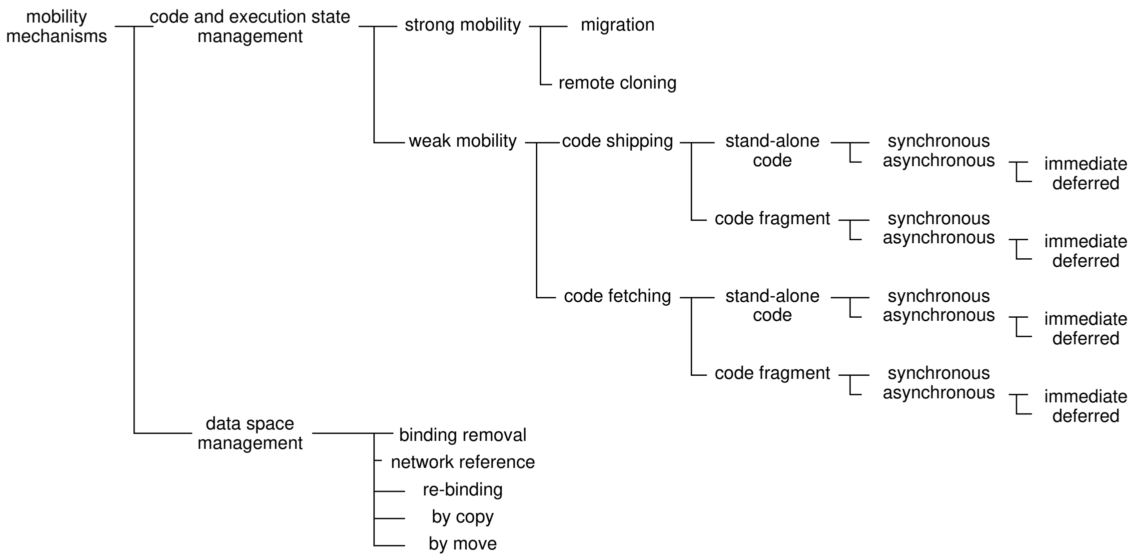


Figure 1.1: Classification of mobility mechanisms [FPV98]

The weakest binding is the binding *by type*. The executing unit needs a resource of a specific type, no matter which identity or value it has. In this case it is sufficient for the data space management to *re-bind*. The problem of *re-binding* is the resource reallocation that may fail if no resource with an appropriate type is available in the new computational environment.

Figure 1.1 illustrates the classification of technologies for mobile code. The different architectures which utilize these mobility mechanisms are described in the following section.

### 1.1.2 Design Paradigms

The common characteristic of architectures deploying mobile code is the distribution of know-how (code) and resources (data) over sites, i.e. representation of location. Another aspect deals with the computational component which triggers the execution of the know-how.

#### Remote Evaluation

In the paradigm of *remote evaluation* a computational component $A$ has the know-how in order to execute the task, but it lacks the necessary resources, which are suited on a different site. The computational component $A$ therefore sends the know-how to a computational component $B$, which resides on the same site as the resource, needed to fulfill the task. The computational component $B$ executes the task using the know-how received from $A$. The results of the task are delivered back to $A$.

#### Code on Demand

In the *code on demand* paradigm, the computational component $A$ has local access to the resources, but does not know how to execute the task. Thus, it contacts a computational component $B$ on a different site, which provides the know-how. The component $A$ loads the know-how from $B$ and executes the task locally.

**Mobile Agent**

The *mobile agent* paradigm is applied to a similar scenario as the remote evaluation, where the resources reside on a different site than the computational component $A$. The idea of the *mobile agent* paradigm is to send the entire computational component $A$ to the remote site and not only the know-how. In contrast to the remote evaluation paradigm the *mobile agent* paradigm does not presume a computational component $B$ residing on the same site as the resources for execution of the know-how. The *mobile agent* paradigm differs from the *remote evaluation* and the *code on demand* paradigms concerning the movement of an existing and running computational component.

### 1.1.3 Application Domains

After the presentation of mobility mechanisms and architectures of mobile code, this section gives an overview of application domains for mobile code.

A potential application of the *mobile agent* paradigm is the scenario of distributed information retrieval, where a small subset of data must be selected from a big superset of data distributed over a set of hosts. In all traditional client-server architecture the whole superset must be moved over the network to the computational component, that executes the selection. When deploying *mobile agents* the computation is executed close to the information base, which avoids the communication over the network for the selection. The *mobile agent* transports over the network only the small subset of data selected out of the information base.

A wide spread form of mobile code is the enhancement of electronic documents for interactivity and presentation effects, like web pages or e-mail. It enables the local execution of programs that are associated with the document. For instance, user inputs can be locally evaluated and improve the interactivity of the document. The usual paradigm used for this scenario is *code on demand*. E.g. WWW technologies and Java applets offer the necessary mechanism for active documents.

Another scenario which is interesting for the deployment of mobile code is the remote control of devices, e.g. network management. Especially the monitoring where the status of devices are checked by continuous polling may generate considerable network traffic and may cause other problems in a traditional client-server architecture. The alternative is to move the monitoring functionality to the devices and to fire events reporting about the state of the devices.

In the area of work flow management systems activities can be mapped on mobile components. The mobile components circulate through all participating entities which are involved in the work flow. The mobile components which encapsulate documents can control access rights to the document and the revision status of the document.

Active networks may be considered as another application of mobile code [TSS+97]. It is used to manage the network through the traffic itself. There exist basically two approaches: the programmable switch and the capsule. The programmable switch conforms with the *code on demand* approach, where the node loads executable code for extended functionality. The capsule approach embeds the executable code into the packets. The embedded code is executed at every node which is passed by the packet.

In the scope of electronic commerce mobile code helps to customize the behavior of participating parties by encapsulation of protocols. It also supports moving *mobile agents* in order to operate close to the information base or to other *mobile agents* participating at the negotiations, e.g. in a virtual marketplace.

## 1.2 Motivation for Adaptation

### 1.2.1 Situation

Mobile code is faced with the problem of heterogeneous hosts. Hosts differ in hard- and software configuration. A first approach is a common base on every host, that provides a homogeneous computational environment to the mobile code, e.g. Java [LY99]. The homogeneous interface of the computational environment is achieved by abstraction from the environment-dependent details. For instance, Java abstracts the file system and offers a platform independent API for accessing its resources. Because platforms strongly differ it is not possible to abstract the entire functionality, which constrains the abstraction to a subset of what is available. The approach of abstraction works fine as long as the highest common denominator is sufficient to the application. If the demands of the application can not be satisfied by the platform independent computational environment, it is necessary to add environment dependent implementations to the application. The application has to decide when to use the right implementation.

### 1.2.2 Static Customization

The general way to do that might be an if-then-else construct with conditional branches, which are interchangeably executed depending on the environment. The common computational environment is still necessary for the bootstrapping of the mobile code. This solution is denoted in this work as *static customization.*

Under certain conditions this solution is not very efficient. Code is moved over the network, which is perhaps used rarely. If a mobile agent for example has to execute its task on hundreds of hosts and ten of them afford a special implementation, which might be rather big, it is not efficient to move the big implementation along the whole route.

### 1.2.3 Dynamic Adaptation

The intention is to enable the mobile code to move with a small core part which is environment independent through the net and to add dynamically, i.e. without termination, environment dependent implementation for the particular environments only. The environment is assumed to be static for the runtime of the mobile code on a host. A change of the environment occurs only if

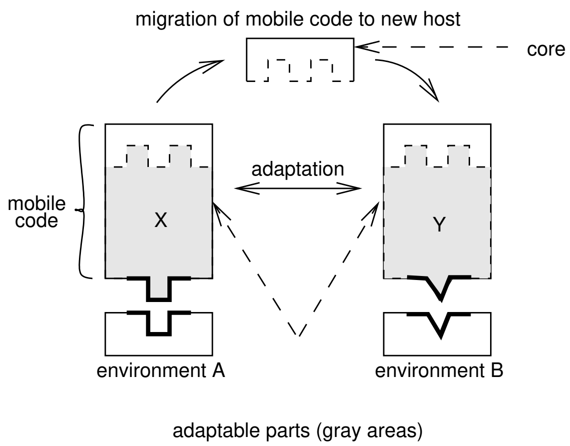


Figure 1.2: Dynamic adaptation overview

the mobile code moves to a different host. The decision which implementation must be used is done without a prior knowledge of the environment of the host.

The mobile code is split into an environment-independent non-adaptable core part and an environment-dependent adaptable part. With dynamic adaptation only code is moved over the network when it is actually needed.

The input of dynamic adaptation is a set of environment dependent implementations $(X, Y)$, a mobile core application and an environment. The result of dynamic adaptation is the selection of the right implementation for the environment and the linking of the selected implementation into the core application. This is also depicted in figure 1.2.

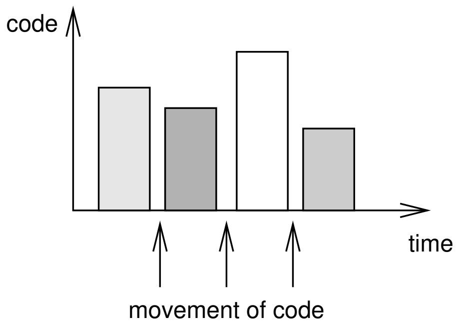


Figure 1.3: Event and object of dynamic adaptation

The trigger for dynamic adaptation is the movement of code. If the mobile code moves to a new host, the dynamic adaptation procedure is initiated. (s. fig. 1.3).

The objects of dynamic adaptation are code fragments, which are exchanged (s. fig. 1.3).

## 1.3 Scenarios

The deployment of mobile code in the area of network management is a promising approach [FKK99], [GFP99],[BPW98], [MKKM98]. As described at the beginning of the chapter, today's computer networks are heterogeneous and thus network management has to deal with managing heterogeneous devices. Even when using platform-independent technologies, e.g. Java, the device properties can differ so strongly that it is necessary to apply implementations which are tailored for specific classes of devices only and packed altogether to support a broad spectrum of environments. For instance, the configuration of network elements by Java technology affords to deal with a variety of device sizes apart from producer specific interfaces. Enhanced PC's serving as routers offer the full functionality as provided by a standard JVM, in contrast to specific router devices which do not offer the execution of Java programs. Though small ipEngines [Bri98] support common operating systems and the execution of Java programs in a JVM, the available resources in terms of memory are very limited and special version of Java programs are necessary.

Another common scenario can be found in the world of Java applets. Different configurations of browsers, which provide the JVM for the applet, support different Java versions and different set of pre-installed Java packages. This leads to incompatible Java applets, if the user does not interact and add necessary Java packages, e.g. Swing packages. The instrumentation of an applet with all necessary packages, which could miss for the execution, is undesirable because of long loading times and inefficient usage of bandwidth. Dynamic adaptation offers an alternative by small applets which can be loaded and executed by every browser, and triggers the loading of additional packages or classes if needed. For example, if swing packages are missing it can switch to a GUI based on the traditional Java AWT.

A third scenario which also belongs to the field of network management, deals with the configuration of applications by mobile code. A set of applications is installed on hosts using different operating systems and cpu architectures. The configuration of the application depends on the system parameters of the host. In a mobile code architecture which does not rely on adaptation, it is either necessary to know the exact hard- and software configuration of every host, which affords complex administration and maintenance, or code for all possible configuration is packed together and sent to the hosts, which means long latency and network traffic that could be avoided. The configuration of a network browser which is selected as the application example for such a scenario for illustration purposes in this work. It is also taken as example application of the prototype implementation, which will be described in chapter 2.

## 1.4 Problem Statement

The goal of this thesis is a concept for the development of mobile code using dynamic adaptation. The concept provides guidance for the programmer to deploy adaptation to its application and it offers a framework architecture which supports the detection of the environment. The framework includes a description and selection mechanism for choosing the right implementation in an environment. Furthermore the framework offers the technology for loading environment dependent implementations from a repository. The presented concepts are implemented as a prototype in order to show that dynamic adaptation enhances mobile code as described in this chapter.

The following chapter gives an overview over the state-of-the-art in adaptation and evaluates various concepts as potential solutions for dynamic adaptation of mobile code. For the evaluation of the known adaptation mechanisms a list of requirements is set up. In chapter 3 the proposed methodology will be presented and the implementation of the proposed methodology. Chapter 4 evaluates the proposed methodology and the implementation according to the requirements as set up in chapter 2. The conclusion of this work and ideas for future work are the contents of the chapter 5.

# Chapter 2

# State-of-the-Art in Adaptation

This chapter gives a survey over a some existing adaptation techniques. In literature the term adaptation is used to express different things. Therefore, the chapter starts with a classification of the different meanings of adaptation. It continues with the definition of requirements for dynamic adaptation as addressed by this work. The chapter will be concluded with a survey of known adaptation techniques towards their applicability for the problems of dynamic adaptation as considered in the scope of this thesis.

## 2.1 Adaptation in Literature

Adaptation techniques as found in literature are used within different contexts. Thus, the objectives, which are addressed, are different and the methodologies differ in order to meet the targets.

### 2.1.1 Static Adaptation

The reuse of code is a field where adaptation is applied. This kind of adaptation is here denoted as *static adaptation*. One of the benefits of component based software engineering (CBSE) is the reuse of existing code respective components. The goal is to reduce programming to the composition of components. Even if components are available for every functionality, it is probable that not every component fits together with another component or into an application. The reasons for that can e.g. be syntactical incompatibility or semantic differences of the interfaces. In order to use incompatible components static adaptation can be used to modify the incompatible parts of code in such a way that they fit together.

Static adaptation differs from dynamic adaptation – as defined in section 1.2.3 – by two criteria. One criterion is the point of time when adaptation is executed. For static adaptation it is sufficient when the adapted code is available at compilation time and remains unchanged during the runtime of the application. In contrast dynamic adaptation is executed without termination of the application.

The other criterion is the nature of the adaptation function. Static adaptation gets a component $C$ as input. The output is the modified component $C'$ which fits into the designated application (s. figure 2.1). This differs from dynamic adaptation (s. section 1.2.3), where adaptation means the exchange of components, e.g. $X$ and $Y$ (s. figure 1.2), depending on the environment.

### 2.1.2 Continuous Adaptation

For some applications it is important to be able to react to changes of the resources in order to provide a reasonable service even with low resources. For instance, multimedia applications which use unreliable connections, e.g. wireless communication, Internet, must modify the representation of data according to the conditions of the network in order to deliver usable results. Changes of the resource conditions may occur without following a certain pattern or any other regularity.

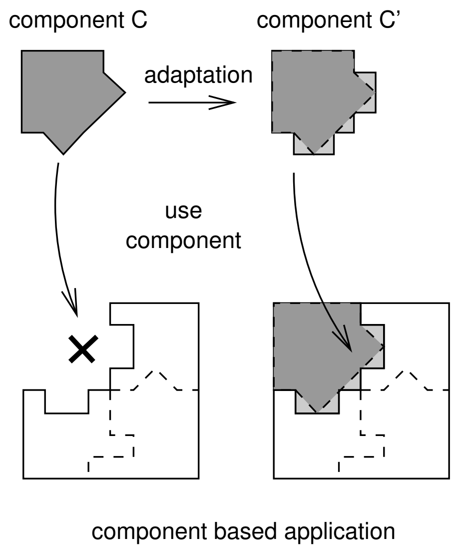


Figure 2.1: Static adaptation

The modification of the application is done by tuning of parameters. Such modifications are here denoted as *continuous adaptation*. The triggers for the continuous adaptation are not discrete events as for dynamic adaptation, but continuously changing conditions of resources.

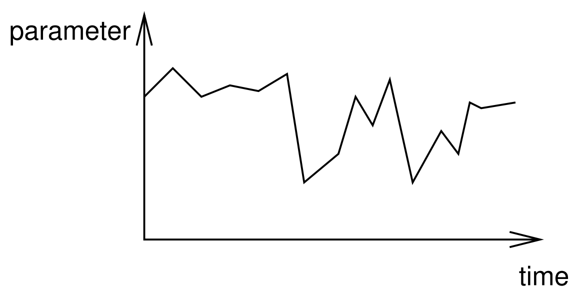


Figure 2.2: Continuous adaptation

For continuous adaptation the resources are monitored and the adaptation process initiated as the resource conditions change. The result of the continuous adaptation is the modification of parameters (s. figure 2.2).

Though static adaptation and continuous adaptation differ concerning the objective and in the realization they share some problems with dynamic adaptation and will therefore be evaluated in this chapter. For the evaluation a list of requirements is specified in the following section.

## 2.2 Requirements

From the goal of this thesis (s. section 1.4) a generic and abstract architecture with main elements as shown in figure 1.2 can be derived. The term *adaptable parts* (grey areas in fig. 1.2) is a general expression and stands for components, modules, objects, methods, byte code, etc. depending on the granularity of the adaptation mechanism and the software technology of the mobile code (e.g. component model, object oriented). These adaptable parts docked into the *core* together form the whole mobile code. The core consists of non-adaptable parts and provides the interface for the adaptable parts. It is used to bootstrap the adaptation mechanism for a particular piece of adaptation software. In the following *adaptable methods* are used to describe methods which build adaptable parts themselves or are contained in adaptable parts, e.g. classes.

From this architecture and the objective target to support application programmers to develop adaptable mobile code following classes of requirements can be derived:

1. requirements concerning the overhead effort necessary for programming adaptable mobile code that fits into a generic adaptation framework
2. requirements concerning the interaction between core and adaptation framework
3. requirements concerning the costs of running adaptable mobile code (bandwidth, CPU usage, runtime, etc.)

These classes of requirements are explained in more detail in the following subsections. Each requirement is labeled with an acronym *R<requirement number>* for further reference in the survey. A summary of all requirements can be found in table 2.1.

### 2.2.1 Programming Overhead: R1-R4

The programming overhead is a very critical point in terms of convincing a programmer to use dynamic adaptation instead of traditional solutions, e.g. conditional branches (s. section 1.2), which covers all known environments.

R1 The integration of the adaptable parts into the core should be possible without any meta language, specifying any interfaces or dependencies of adaptable parts. The same programming language which is used for the rest of the mobile code should be applied for the adaptable parts, to specify their environmental conditions and to put together the whole mobile code.

It does not forbid the deployment of different programming languages in the application, but the code fragments where adaptable parts are used, should not imply special tags or other markers that are not conform with the syntax of the language of the surrounding program code.

The reason for this rule is the kind of compilers, syntax checkers, visualization tools, etc. which are used to develop the mobile code. Only tools for common languages should be necessary and reduce the effort for installing and maintaining the programming environment.

R2 The second requirement improving the convenience for the application programmer is a qualitative requirement: minimal increase of "lines of code". This demand emerges from the direct comparison of an implementation using dynamic adaptation and an traditional implementation providing support for different environments (s. section 1.2). Another unit of measurement could be the number of classes, if an object-oriented (OO) language is used.

R3 Support for easy maintenance can be achieved by a compact and integrated structure of the additional information needed for adaptation. Such information contains the description of environments and the environmental condition of adaptable parts.

R4 An assistance to the application programmer and an increase of stability of the mobile code is the indication and elimination of errors due to adaptation already at compilation time. Following conditions should be ensured by compilation:

- fitting interface between core and adaptable parts
- type safe invocation of adaptable methods
- validity of environmental condition for adaptable parts

The last requirement is an ill-defined requirement because it depends on the architecture of the adaptation mechanism which determines the kind of faults that can occur and therefore should be checked at compilation.

These requirements may not guarantee a more sophisticated way to program mobile code which must handle different environments but they guarantee at least, that the effort for using dynamic adaptation is not noticeable higher than for the traditional method (s. section 1.2). Beside the aspect of pure programming techniques it is also important to look at the interaction between the core and the adaptation mechanism.

### 2.2.2 Interaction between Core and Adaptation Mechanism: R5-R10

If mobile code uses adaptation, one interesting question is: How much does the core (s. section 2.2) have to know about the fact that adaptation is used? The most desirable answer would be: Nothing. That means that the core does not have to worry e.g. when to trigger adaptation, how to handle any faults during adaptation or to care for linking any adaptable parts: full transparency of adaptation. The following requirements build a basic set to get towards full transparency.

R5 As defined in section 1.2.3 the mobile code must not be terminated for the adaptation.

R6 Another requirement in order to get more transparency is the demand for minimal impact on the characteristics of the mobile code. E.g. if the adaptation mechanism would strongly rely on a central server, it would loose its autonomy, an important characteristic of mobile code.

R7 The initiation of the adaptation should not be task of the core. The adaptation mechanism must be able to notice itself when it is necessary to reconfigure the mobile code.

R8 This requirement which may overlap with the former is the demand for linking adaptable parts without participation by the core. If the mobile code is being adapted, parts perhaps defined as modules, components, objects or other other kind of units must be exchanged. From where they are loaded, e.g. over the network or from a specific directory, and how they are linked into the running core should not be mixed up with the task of e.g. performing configuration management and executing a specific task.

Method invocation on adaptable mobile code should not differ from standard method calls.

R9 There should be no special syntax required for calling a method offered by an adaptable part.

R10 Also the semantic (`R10`) of the method must simulate the invocation of a non-adaptable method. In distributed environments, like CORBA [OH98] or Java RMI [Sun99], a similar problem occurs. The handling of remote objects is hidden to the application by local stubs, which encapsulate the communication with the server.

The requirement of the standard syntax used for adaptable method calls overlaps particularly with requirements `R1-R4`, which concern the programming language and syntax for implementing adaptable mobile code.

After considerations concerning programming effort and transparency the following topic specifies some demands concerning the influence of adaptation on the performance of the mobile code.

### 2.2.3 Runtime Overhead: R11-R13

One of the advantages of adaptation is the potentially reduced size of code which must be shipped over the network. This gain of bandwidth and loading time must not be wasted by an expensive adaptation mechanism.

R11 Therefore a fundamental requirement is to demand that the size of the binary resources of the core and of an adaptable part (worst case: the biggest) including the adaptation mechanism is smaller than the binaries of a comparable monolithic application. Obviously, this needs to be considered in relation to the overall size and to the execution time of the code. This requirement supports the argument of saved bandwidth when using adaptation.

| requirement class | requirement | acronym |
|---|---|---|
| programming effort | no meta language for adaptation | `R1` |
| | source code overhead | `R2` |
| | integrated solution | `R3` |
| | high error detection during compilation | `R4` |
| interaction between core and adaptation | adaptation without termination | `R5` |
| | minimal effect on mobile code semantics | `R6` |
| | automatic initiation of adaptation | `R7` |
| | automatic linking of parts | `R8` |
| | standard syntax for adaptable methods | `R9` |
| | transparent method call | `R10` |
| runtime overhead | smaller size than monolithic application | `R11` |
| | low time for re-configuration | `R12` |
| | low time for invocation of adaptable methods | `R13` |

Table 2.1: Summary of requirements

R12 The second requirement enforcing minimal overhead is fast loading of adaptable parts and re-configuration of the mobile code.

R13 Considerations on runtime overhead must also include the time needed for the invocation of adaptable methods. If this is noticeably high, e.g. in relation to the runtime of the method, the adaptation mechanism has a negative impact on the performance of the mobile code.

### 2.2.4 Deployment of Requirements

The listed requirements differ in the way they can be deployed for an evaluation of adaptation mechanisms. Some requirements allow a well-defined validation, e.g. `R1` - yes it does use a meta language or no, it does not-, others help to compare two concepts by doing measurements, e.g. `R11`-`R13`, but others allow only a qualitative evaluation of a concept, e.g. `R4`, which requires *high error detection during runtime*. Such a requirement can only be deployed by discussing the concept under the aspect of the requirement, but an exact judgment is not possible.

## 2.3 Survey of techniques

As mentioned in section 2.1 the known adaptation mechanisms can be classified into dynamic adaptation, static adaptation and continuous adaptation. Though most of the found mechanisms are not mechanisms for dynamic adaptation, it is worth to evaluate some of them and to analyse if parts can be used for dynamic adaptation.

### 2.3.1 Static Adaptation

In [Hei99] a collection and evaluation of component adaptation mechanisms is presented. They can all be classified as members of the static adaptation category. The main problem that static adaptation mechanisms are not usable for dynamic adaptation is the adaptation at compilation time.

#### Binary Component Adaptation

The *Binary Component Adaptation* (BCA) [KH98] shifts parts of the adaptation from the compilation time towards the runtime.

The BCA mechanism applies adaptation to classes. Attributes, methods and modifiers of classes can be removed, added or changed without needing to access the source code. The modifications are specified in an *adaptation file* which is compiled into a binary *delta file*. The delta file is then being merged with the original binary resources during loading and adapted binary code is produced. Keller et al. used the Java technology for the implementation of their prototype. The heart of the BCA technique is a modified class loader. The original procedure of class loading

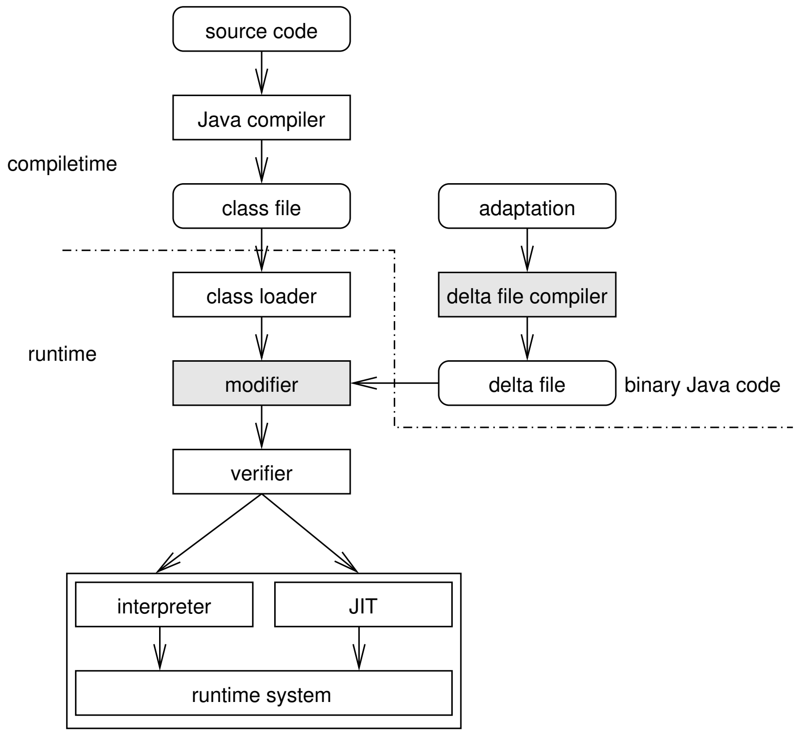


Figure 2.3: Procedure of class loading in BCA

[LY99] is extended by a *modifier* and a *delta file compiler* (s. fig. 2.3). In the original procedure a class file is loaded by the class loader and translated into an internal representation. The internal representation is then checked by the verifier to guarantee that no JVM rules are violated. In the modified procedure the BCA modifier is inserted between class loader and verifier. The modifier takes a class in the internal representation from the class loader and adapts the class according to the specification in the binary delta file. The delta file was generated from the adaptation file. In the adaptation file the programmer can define the modifications on a class in a syntax similar to Java. For example if the programmer wants to add an interface `Printable` and an implementation of a method `print(String msg)` as defined in the interface to a class `A`, the adaptation file would look as follows:

```
delta class A {
      add interface Printable;
      add method public void print(String msg){
          System.out.println(msg);
      }
}
```

After the modification of the loaded class file it is passed to the verifier and executed by the interpreter or a just-in-time compiler (JIT).

Though BCA offers adaptation not only at compilation time, but also at load time – as imposed by `R5` – in its original from it is not suitable for dynamic adaptation. It assumes that there is at maximum one delta file for a class file and does not provide any functionality to perform a environment specific selection of a delta file out of a set of delta files.

### Load-Time Adaptation

Load-Time Adaptation (LTA) [DH88] uses a similar approach as BCA. The class files are modified at load time. The specialty of LTA is the point where the loading process is interrupted and adaptation is executed. BCA is implemented as a customized class loader, but LTA acts on a lower level in order to keep the loading process between virtual machine and application untouched. The intention is to redirect the operations on the class file, e.g. `read()`, `open()`, `close()`.

The redirection of system calls is neglected because modifications of the operating system are hard to port to various platforms. The solution as proposed by [DH88] is to modify the dynamic library, which provides the functionality for operations on the file system. The modified library accesses the file system via system calls and performs an adaptation of the class files if necessary. This approach hides the adaptation process to the application and to the virtual machine, but supports porting the modified dynamic libraries to various platforms, because it uses only non-OS dependent functionality.

This LTA adaptation mechanism suffers on the same lack as BCA: the missing support for an environment dependent adaptation.

### MetaJava

The adaptation of runtime mechanisms is discussed in [GK97] and a platform for adaptable operating system mechanisms, called *MetaJava*, is presented. MetaJava provides *reflection* for controlling the execution environment of an application. The behavioral reflection is different from the Java Reflection which is denoted in [GK97] as structural reflection. Another aspect of MetaJava is the separation of functional code and non-functional code, called *metaprogramming*. The functional code, which concerns the computation of the application, resides in the *base level* and the non-functional code, which controls objects in the base level, resides in the *meta level* (s. fig. 2.4).

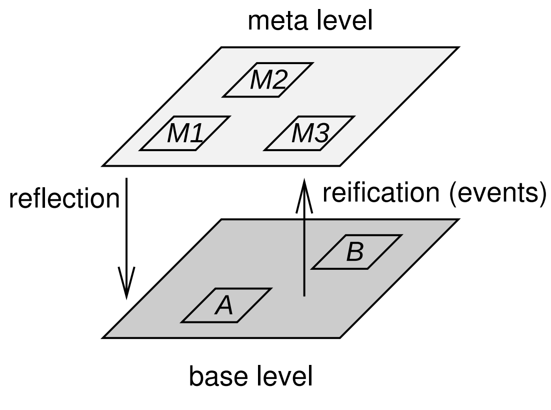


Figure 2.4: Behavioral reflection and reification

Whereas in traditional systems ad hoc extensions of the programming language support multiple threading, distribution, fault tolerance, mobile objects, etc., MetaJava hides such functionality in the meta level.

For instance, Java supports the programming of multi threads or Remote Method Invocation, but the programming is done on the same level as the programming of the application task which might be independent from the thread management or object distribution.

For triggering the meta level *reification* is necessary. Reification is defined in [GK97] as “the process of making something explicit that is normally not part of the language or programming model". The reification is realized using events that are raised by the base level and delivered to the meta level. For using the meta level a meta object must be attached to an object in the base level. Meta objects can be attached to references, objects and classes.

An application for MetaJava is for example a special type of remote method invocation where instead of the Java build-in RMI package a proprietary mechanism implements remote method invocation over a proprietary protocol stack.

If a method of a remote object is called, an event of the local method call of the proxy object is delivered to the attached meta object. The meta object then propagates the method invocation to the server.

In the context of adaptation an event for class loading might be used in order to initiate the adaptation mechanism and to select the right class for the current environment.

For the integration of MetaJava a modified Java Virtual Machine (JVM), the *MetaJava Java Virtual Machine* (MJVM), is used. The MJVM uses the same class file format and byte code set as the JVM, but provides a *meta-level interface* (MLI) permitting the meta objects to access the internal state of the virtual machine.

This implementation implies that the deployment of MetaJava based applications is constrained to hosts on which a MJVM is installed.

MetaJava could provide the architecture for a dynamic adaptation mechanism. The adaptation is executed in the meta level hidden to the application. But the constraint of a MJVM contradicts especially in particular requirement `R6` because the assumption to install everywhere a MJVM limits the radius of the mobile code to such configured environments. The dynamic adaptation would limit the mobility of the code. Therefore MetaJava seems not to be the best technology for dynamic adaptation.

### 2.3.2 Dynamic Adaptation

**LEAD**

In [AW97] a language for dynamically adaptable applications is presented. LEAD is a methodology of the field of dynamic adaptation. The granularity of LEAD is a procedure. LEAD enables applications to select the right implementation according to the current environment from a set of available procedure implementations for various environments. In LEAD such an adaptable procedure is called *generic procedure call* and an environment specific implementation for this *generic procedure call* is named *method.* The static (non-adaptable) part of a program is the *base-level* which calls generic procedures in the *meta-level.* LEAD consists of the following parts:

1. a new programming language, which supports the definition of generic procedure calls and associated methods

2. a runtime system, which selects the appropriate method for a generic procedure call

If the runtime system in the meta-level receives a generic procedure call from the application, it obtains the information about the current environment from the OS, evaluates the information based on a built-in adaptation strategy and selects the suitable method. The method is sent back to the application and executed on the base-level.

The runtime system can detect changes in the environment and informs the application by asynchronous events. Generic procedure calls are associated with those events and enable the application to react on a changed environment. The environment is described by *environmental elements* which are an abstraction of the environment. LEAD provides a fixed set of basic environmental elements:

| environmental element | property |
|---|---|
| network | represents network connectivity |
| memory | free memory size |
| display | display size |
| power | power source type |
| battery | battery remainder |

A function is associated with each environmental element and is executed by the runtime system determining the state of the environmental element. Based on the built-in elements new environmental elements can be constructed on a higher abstraction level.

The definition of a method contains a list of conditions referring to states of environmental elements. The list is evaluated by the runtime system and the result of the evaluation determines the selection of the method.

How the list of conditions is evaluated is defined by the *adaptation strategy*. The goal of the evaluation is to determine a suitable method for the environment. By default the simple strategy is applied which evaluates the condition list with *AND* and *OR*s. If this strategy fails, the *priority strategy* is applied, which evaluates the conditions according to the priorities associated with them.

The LEAD approach can not fulfill two requirements defined in section 2.2. The requirement `R1` which interdicts a meta language for dynamic adaptation is violated by LEAD. LEAD presents an extension of *Scheme* in order to define adaptable procedures. This would involve a new programming language into the development of an application for using adaptation.

The second violation of a requirement is a consequence of the first. The definition of adaptable procedures as string variables in *Scheme*, prevents that the adaptable methods are syntax-checked during compilation time and runtime errors may occur during their interpretation at runtime. The purpose of `R4` is to avoid such a situation.

Due to the listed reasons it is not desirable to use LEAD for dynamic adaptation.

### 2.3.3 Continuous Adaptation

Continuous adaptation is merely interesting for mobile devices, which are faced with a continuous changing environment. The main focus of continuous adaptation is the detection of the current environment and the propagation of environment changes to the application. The actual modification of the application's behavior is mostly limited to resetting the parameters.

In [STW92] the environment dependent parameters are managed by *dynamic environment servers*. If an application is interested in the parameters of a dynamic environment server, it must subscribe to the server. For the communication between the application and the server RPC is used. The application can retrieve parameters from the server, but the server can also notify the application by callbacks about changes in the environment.

[Nob00] presents a scenario, where a small personal digital assistant (PDA) guides tourists through a museum or a city delivering multimedia information about touristic attractions. The PDA must handle changing bandwidth and be aware about the tourist's location in order to present the interesting information of the location. The focus of [Nob00] is to decide whether the adaptation is executed by the application, called *laissez-faire application*. If the adaptation is triggered by the operating system, the adaptation is called *application-transparent system*. The application has the knowledge, which amount of resources is necessary, whereas the operating system has an overview of the running applications and can manage the shared resources. The conclusion of [Nob00] is a collaboration of operating system and application, denoted as *application-aware collaboration*. The collaboration is realized as an adaptive decision loop in the application. Inside the loop the application selects the data quality and requests the according resources. The application is informed by the operating system about changes of the resources. After notification the application selects a new data quality and redefines its resource requests.

A similar scenario is presented in [ADOB98]. The work presented in this section deals with different problems involved in manipulating parameters than dynamic adaptation which exchanges executable code. The common object of continuous adaptation and dynamic adaptation is the detection of the environment in order to determine the right adaptation. The information retrieval from the environment is based on application specific sensors, like active badges [WHFG92], and not on generic properties of the hard- and software configuration, as targeted by this work.

## 2.4 Conclusion

Besides dynamic adaptation as defined in section 1.2.3, static adaptation and continuous adaptation can be found in literature. Both forms have different objectives than dynamic adaptation. The main reasons that methodologies for static adaptation can not be used for dynamic adaptation are:

- the time of adaptation, which is at compilation
- the missing concept for the environment detection

Though continuous adaptation is providing a functionality for inspecting the environment the methodologies are not suitable as a whole because of

- application specific sensors
- modification of parameters instead of code

The consequence of this considerations is the necessity to develop an own methodology integrating the exchange of code during runtime and the detection of the environment flexible for all kinds of properties concerning the hard- and software configuration of a host.

# Chapter 3

# Proposed Methodology of Adaptation

Since the evaluation of chapter 2 concludes that none of the currently known adaptation techniques meet the requirements, a new and specifically tailored methodology will be developed in this chapter. For illustration purposes an example is introduced in the first section, which will be referred to throughout the remaining chapter.

## 3.1 Example: Browser Configuration

The background of the example is a simple configuration management architecture using mobile agents, as a special form of mobile code (s. section 1.1.2), based on Java technology for executing tasks in intranets.

The term mobile agent is used to denote executable code which migrates to different execution environments to perform its tasks. The work does not rely on other specific aspects of mobile agents for this particular application.

Every host which supports the management architecture has an agent system running serving as a common platform for agents. Objectspace Voyager [Obj00] is used as mobile agent platform. The primitive agents are initialized with a list of hosts and move from one host in the list to the next to locally execute their task. After execution of the task on the last host, the agents terminate. The configuration procedure must set the size of the disk cache and the memory cache of the browser. The setting of these parameters must be system dependent in order to get the optimal cache size, hence the opportunity for code adaptation to fit to local circumstances.

The example agents are programmed to set the size of the disk and the memory cache to a reasonable percentage, e.g. 10%, of the free disk space for the disk cache and of the physical memory for the memory cache.

The agents must support various CPU architectures and operating system configurations, e.g. Windows NT/x86, Linux/x86, AIX/PowerPC, etc. and different web browser types, e.g. Netscape Communicator, Microsoft Internet Explorer, Lynx. The access to system information, like physical memory size and free disk space, is in most cases not provided by Java functions and must be implemented in platform dependent functions. Even if the configuration files for the browsers are in text format and accessible by platform independent Java APIs, e.g. Netscape, Lynx, there are still the syntactical differences between the configuration files of different browsers.

The support of different platforms and applications and the lack of standardized interfaces for this task forces the agent programmer to instrument the agent with various implementation variants for various platforms and browser types. As pointed out earlier, this could result in an implementation that uses static customization (s. section 1.2.2) for choosing between different forms of behavior. To motivate the adaptation framework presented in this work, a more advanced solution is however assumed in the following based on code adaptation.

The collection of system information and the configuration of the browser is e.g. done in the function `configBrowserCache(int,int)` that may look as presented in in figure 3.1, for a non-adaptable program.

```
public void configBrowserCache(int diskCachePercent, int memCachePercent){
    int memSize = m_memory.getPhysicalMemorySize();
    int freeDiskSpace = m_harddisk.getFreeDiskSpace();
    int totalDiskSpace = m_harddisk.getTotalDiskSpace();

    m_configuration.setDiskCache(diskCachePercent,
                                 freeDiskSpace);

    int cacheMemSize = (memSize * memCachePercent)/100;
    m_configuration.setMemoryCache(cacheMemSize);
}
```

Figure 3.1: Non-adaptable version of `configBrowserCache`

Assuming that methods implemented by the object `m_memory` of the class `Memory` reads the size of the physical memory, the object `m_harddisk` of the class `Harddisk` reads the size of free disk space and the size of total disk space. The configuration of the browser is done by `m_configuration` of class `Configuration`. The objects `m_memory`, `m_harddisk` and `m_configuration` are member variables and therefore marked by convention with the prefix `m_`.

The classes `Memory`, `Harddisk` and `Configuration` or some of their methods must be implemented in different ways and using different technologies. E.g. for Linux the `getPhysicalMemorySize()` functions can be implemented in pure Java reading the text file `meminfo` in the `/proc` file system, in contrast to a Windows environment, where either a native diagnostic tool or a C function of the Win32 API must be used instead. The C functions can be embedded into the Java code by using the *Java Native Interface* (JNI) [Lia99].

Before the various aspects of adaptation are discussed, a short overview of adaptation will be presented in the following section.

## 3.2 Overview

The main elements which are involved in adaptation are the environment, the core with its adaptable parts and the adaptation framework (s. fig. 3.2). The environment are the hard- and software configuration of a host or any other *static* properties. *Static* properties means in this context, that the properties do not change during the runtime of the mobile code. Examples for such properties might be the CPU architecture, operating system, default web browser, etc..

The core is the non-adaptable part of the mobile code. Note that its existence is a necessary boundary condition, since adaptation has to start from a well-defined state. The non-adaptable part must move to every host on which the mobile code will run and form the minimal footprint of executable code which must be transfered over the network. The adaptable parts are implementations of equivalent tasks, but suitable for different environments. The core and the adaptable parts are designed by the application programmer (shaded in grey in fig. 3.2).

An adaptation framework which is application independent provides a mechanism to determine the appropriate adaptable parts for the current environment, referred as *context awareness*, to load the adaptable parts over the network from a repository and to link the parts into the core, denoted as *reconfiguration*, which is the contents of the following section.

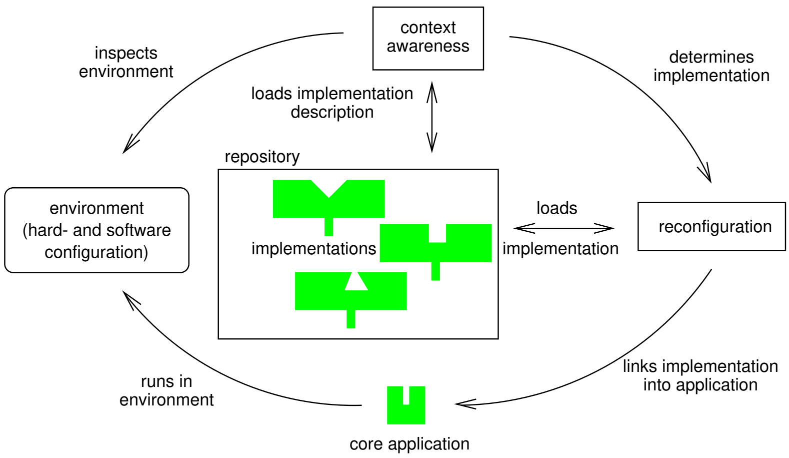


Figure 3.2: Elements/components involved in adaptation

## 3.3 Reconfiguration

The reconfiguration loads executable code and links it into the core. The core can switch between various implementations without termination, (s. requirement `R5` in chapter 2, table 2.1). The linking should be done at a high level of transparency to the core (`R6-10`). From these requirements a design pattern [Gra98] can be derived which must be followed by application programmers in order to do dynamic linking of Java classes. The design pattern will be presented in the first part of this section. The second part of the section deals with the framework offered to the application programmer.

### 3.3.1 Design Pattern

Adaptation links parts, which export the same semantic definition to other parts of the mobile code, but differ in their implementations. This polymorphism can be simulated in the OO world in general and by the Java language in particular in at least two ways:

- *generalization* by superclasses
- *abstraction* by interfaces

In the example the generalization might provide a superclass `Memory` and adaptor classes `Memory_AIX_PPC`, `Memory_WINNT_X86` and `Memory_LINUX_X86` (s. figure 3.3). The specific class example refers to the configuration scenario, introduced in section 3.1.

If a superclass is used to realize polymorphism, functionalities which are common for all environments must be implemented only once in the superclass and can be inherited by the implementation classes. This approach however is only possible in OO languages which support multiple inheritance. If the language supports the single inheritance paradigm, e.g. Java, the use of generalization would constrain the programmer to use a single superclass for all implementations of e.g. `Memory`. For instance, if the application programmer intends to use a class `SystemResources_LINUX`, which supports access to system information, it is not possible to inherit functionality from `SystemResources_LINUX` for the implementation class `Memory_LINUX_X86`, because all implementation classes must inherit the superclass `Memory`.

The abstraction by interfaces offers an alternative to the abstraction by superclasses (s. figure 3.4). It allows the implementation classes to be subclasses of other classes than the one needed for adaptation. For adaptation it is necessary that all implementation classes of an implementation group export the same behaviour to other classes. If this is realized by an interface, the implementation classes are not restricted in the case of a programming language allowing only single

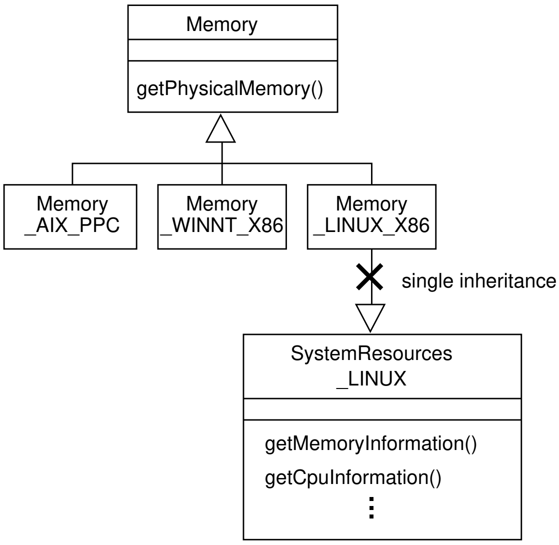


Figure 3.3: Design pattern with superclass and considering Java single inheritance

inheritance, e.g. Java. Figure 3.4 shows the same secnario as figure 3.3, where the implementation class `Memory_LINUX_X86` wants to inherit functionality from the class `SystemResources_LINUX`. In this case the inheritance is also possible for single inheritance because `Memory_LINUX_X86` implements an interface in order to be used as an implementation class and is not part of an inheritance hierarchy due to the adaptation mechanism.

The design pattern for the proposed methodology is based on this concept. Before the concept is described in more detail the necessary terminology which is used within the scope of this work will be defined by the example of the classes for the browser configuration agent. The functionality of the `Memory` class is described in an interface `IMemory`, called *functionality interface*. Implementations of `IMemory` for different environments, called *implementation classes*, export all the same functionality to other parts of the mobile code, but their implementation is suitable for different environments.

Note that much of this is going on in normal modeling as well. However this approach provides in addition the automatic composition at runtime of particular branches in the inheritance relationship according to environmental conditions. Other branches can be left out, thereby releasing cost benefits.

In the example the three functionality interfaces `IMemory`, `IHarddisk`, `IConfiguration` are defined. Table 3.1 shows the implementation classes of the functionality interfaces and a description of their respective environment.

Implementation classes which implement the same functionality interface are called *implementation group*. The name of an implementation group is by convention the common functionality interface of the implementation classes. The implementation classes `Memory_AIX_PPC`, `Memory_WINNT_X86`, `Memory_LINUX_X86` all implement the interface `IMemory` and form an implementation group with the name *IMemory*.

The names of the implementation classes in the example have suffixes that correlate with their desired execution environment. These suffixes are used to give more meaningful names to the classes of the example. E.g. the implementation class `Harddisk_WINNT_X86` could also have the name `Foo`. It is up to the application programmer to name the implementation classes. There is no compulsive correlation between the name of an implementation class and its respective environment.

Figure 3.5 shows two hosts with different configurations. Without changing the interface of the core mobile agent, different implementations (grey shaded) are used fitting to the operating

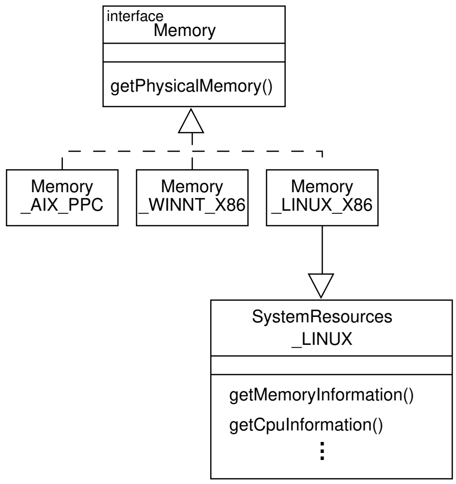


Figure 3.4: Design pattern with interface

| functionality interface | implementation class | environment |
|---|---|---|
| `IMemory` | `Memory_AIX_PPC` | AIX, PowerPC |
| | `Memory_WINNT_X86` | Windows NT, x86 |
| | `Memory_LINUX_X86` | Linux, x86 |
| `IHarddisk` | `Harddisk_AIX_PPC` | AIX, PowerPC |
| | `Harddisk_WINNT_X86` | Windows NT, x86 |
| | `Harddisk_LINUX_X86` | Linux, x86 |
| `IConfiguration` | `Configuration_UX_NETSCAPE` | Unix, Netscape |
| | `Configuration_WINNT_NETSCAPE` | Windows NT, x86, Netscape |
| | `Configuration_UX_LYNX` | Unix, Lynx |
| | `Configuration_WINNT_IEXPLORER` | Windows NT, x86, Internet Explorer |

Table 3.1: Implementation classes for the example

system, CPU architecture and the default browser. The core mobile agent represents the non-adaptable part of the mobile code which handles e.g. the movement in the network or controls the general program flow, e.g. the `configBrowserCache(int,int)` function.

A constraint of this design pattern is the specification of implementation classes by an interface. Such an interface cannot define any member variables of the implementation classes. A method to add member variables to implementation classes is to wrap the implementation classes.

This wrapper classes as part of the interaction between the core, the functionality interfaces and the implementation classes are described in the following section. The interaction between the functionality interface and the implementation class in particular includes the procedure of replacing an implementation class by another class, e.g. to exchange `Memory_AIX_PPC` by `Memory_WINNT_X86` when the agent moves from a host running AIX on a PorwerPC to another host which is running on a x86 CPU with Windows NT as operating system.

An important aspect of reconfiguration is the dynamic linking of the implementation classes into the core. Dynamic linking is supported by the Java core technology [LB98], but in the context of this work, it is also important that the impact of the dynamic linking is as low as possible for the non-adaptable parts of the mobile code. In this section three proposals will be presented. The order in which the concepts are described correlates with the degree of transparency of linking to the mobile code. The section will conclude with an evaluation of the proposed concepts. As a result of the evaluation, one proposal will be chosen for the implementation of the prototype.

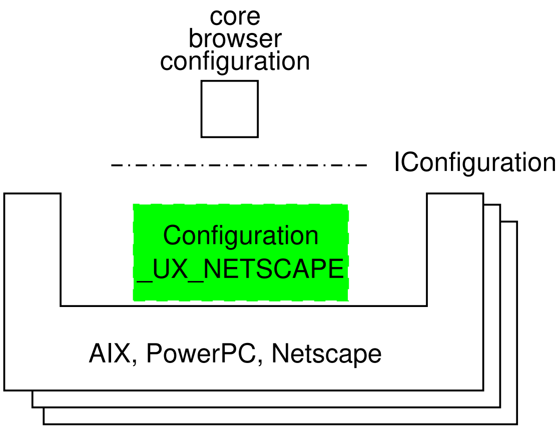


Figure 3.5: Design pattern

### 3.3.2 Explicit Adaptation - Proposal 1

The first concept assumes that there is an object `m_adaptation` in the mobile code available. Its method `loadImplementationClass()` takes the name of the functionality interface and returns the right implementation class. In the example the mobile agent would hold a reference to an object of type `IMemory`. The mobile agent loads if necessary the class `Memory_AIX_PPC`, `Memory_WINNT_X86` or `Memory_LINUX_X86` from the adaptation mechanism specifying the functionality interface `IMemory`. The same applies to the *IHarddisk* and *IConfiguration* implementation group. In this proposal `m_adaptation` acts as an explicit broker which is fully visible for other objects.

```
public class WebClientAgent_Proposal1{
  private transient IMemory m_memory;
  ...
  public void configBrowserCache(int diskCachePercent, int memCachePercent){
    ...
    Class memoryClass = m_adaptation.loadImplementationClass("IMemory");
    m_memory = (IMemory) memoryClass.newInstance();
    int memorySize = m_memory.getPhysicalMemorySize();
    ...
  }
}
```

Figure 3.6: Invocation of `getPhysicalMemorySize()` in proposal 1

Figure 3.6 shows a code example for the invocation of a method of the class `Memory`. Before the method invocation the class must be loaded and instantiated explicitly through the adaptation mechanism `m_adaptation`.

The member variable `m_adaptation` holds a reference to an object of the class `Adaptation`. The class `Adaptation` provides a method `loadImplementationClass(String)` which returns the appropriate class for the environment.

When the agent migrates to a new host, it drops the implementation class and re-creates an implementation class in the new environment. The mobile code does not carry the implementation class to the new environment for the case that the implementation class is not needed on the new host. Figure 3.7 shows this concept where on every host the implementation classes must be explicitly re-created by the mobile agent.

This solution would implicate that adaptation is not transparent for the core:

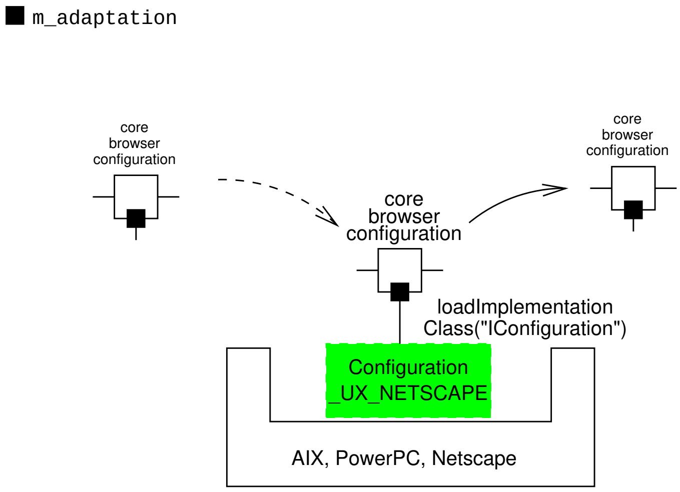


Figure 3.7: Explicit adaptation

1. the application programmer must determine when to adapt classes (disagrees with `R7`)
2. the procedure of class instantiation of adaptable classes differs from the instantiation of non-adaptable classes (disagrees with `R9`)
3. method calls are not checked against type errors at compilation time, which is required by `R4`

A second disadvantage is the loading of the implementation class by specifying the functionality interface during runtime as a string. If a non-existing interface name is specified, the error can only be detected during runtime. This is contradictory the requirement `R4`. An improved level of transparency can be achieved by the following approach.

### 3.3.3 Adaptation Adaptor - Proposal 2

A higher level of transparency can be achieved by wrapping the procedure of adaptation. Instead of using objects of type `IMemory` and loading different implementations, a non-adaptable class wraps the adaptable classes and links them with the core.

Similar to CORBA's IDL stubs [OH98], where stubs hide the marshaling and simulate a local method call, little adaptor classes hide the adaptation from the core, and simulate a method invocation in non-adaptable mobile code. To the functionality interface `IMemory` an adaptor class `IMemory_Adaptor` which implements `IMemory` and is used in the mobile code instead of `IMemory`. The same code as the one above looks with adaptor classes like in figure 3.9.

All interactions between implementation classes and the core are handled by the adaptor classes. They decide when to adapt, hide the dynamic loading mechanism and provide a Java conform class instantiation to the application programmer. The adaptor classes could also be automatically generated at runtime similar to the *proxy class* in Objectspace Voyager [Obj00].

The adaptor holds a reference to an object of the class `Adaptation` (s. section 3.3.2) and loads via that object the right implementation class. The method invocations from the core are delegated by the adaptor to the implementation class.

The disadvantage of this approach is the compilation of the adaptor classes. It involves an additional tool, the adaptor generator, which must be integral into the compilation procedure, and complicates the maintenence of compilation setup, e.g. Makefiles etc. It also does not offer total transparency as desired, because the application programmer uses the adaptor classes with the class name suffix `_Adaptor`.

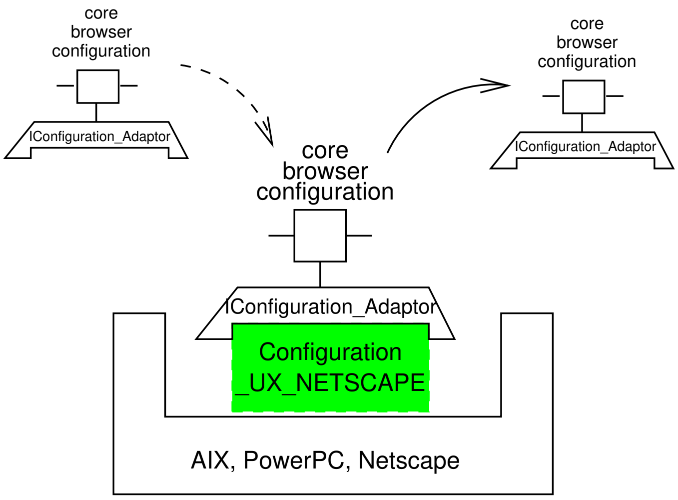


Figure 3.8: Adaptation adaptors

```
public class WebClientAgent_Proposal 2{
  private IMemory m_memory = new IMemory_Adaptor();
  ...
  public void configBrowserCache(int diskCachePercent, int memCachePercent){
     ...
    int memorySize = m_memory.getPhysicalMemorySize();
    ...
  }
}
```

Figure 3.9: Invocation of `getPhysicalMemorySize()` in proposal 2

### 3.3.4 Virtual Class - Proposal 3

Maximum transparency is realized by using a virtual class, e.g. `Memory`, which is used as a representation for the implementation classes and instead of loading the class `Memory` the adaptation mechanism is triggered to load the right implementation class. The virtual classes do not exist as source code or executable code, and serve just as place-holders for the implementation classes.

This might be realized by a modified compiler, which must recognize the class as a virtual class. The virtual class might then be replced by a reference to an adaptor class.

Another approach could be the exchange of the virtual class during runtime by generating adaptors in byte code. Nevertheless there would still be the problem that the compiler needs a code base for the virtual class in order to be able to translate the source code.

Figure 3.11 shows the migration of the mobile agent with the virtual classes and the replacement of implementation classes on the hosts. The proposal of virtual classes would offer a very high level of transparency, but the problem is the compilation of an application that references classes which can not be found by the compiler or the linker.

### 3.3.5 Summary of Reconfiguration

The low gain of transparency achieved by *virtual classes* compared to the *adaptor classes* and the effort for creating a suitable compiling procedure, e.g. by a modified compiler, do not justify the implementation of *virtual classes* for the prototype. As an alternative, the *adaptor classes* are being used for the implementation.

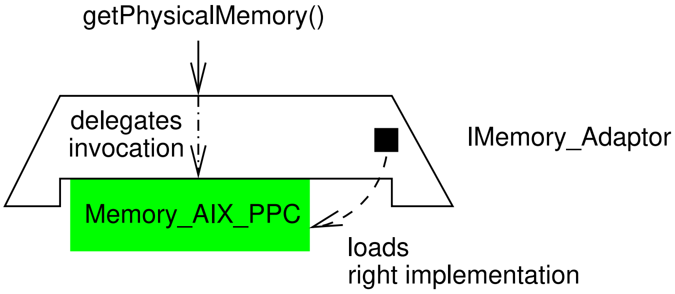


Figure 3.10: Functionality of adaptor

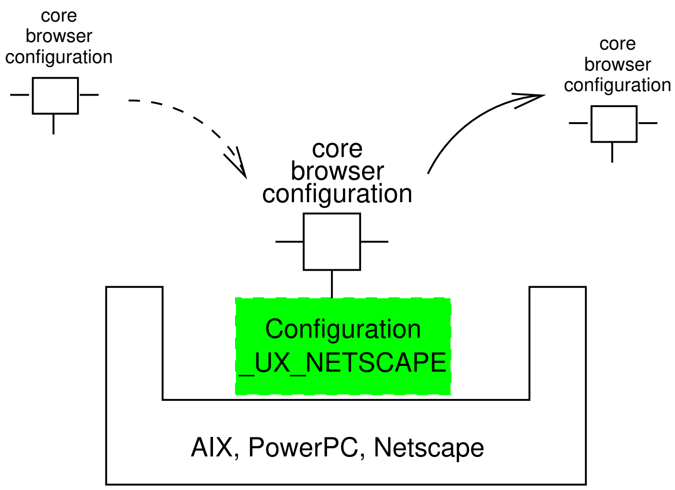


Figure 3.11: Virtual class

A task of the reconfiguration that has not yet been discussed is the loading of the implementation classes. The *loader* acts as intermediate piece between the adaptors and the repository. The implementation classes are loaded from the repository by the loader and handed over to the adaptors where they are linked into the core. The concept of the class loader is a standard concept of the Java technology and supports the implementation of user defined class loaders as needed for the reconfiguration.

The contribution of reconfiguration to adaptation is the design pattern, the concept of adaptor classes and the loader. The design pattern pretends to the application programmer, to define the functionality of the implementation classes in a functionality interface, which is implemented by all implementation classes. E.g. to program an adaptable class that retrieves the information about the total and the free disk space, i.e. `Harddisk` in a non-adaptable version, the programmer defines for an adaptable version the functionality interface `IHarddisk`, and the implementation classes `Harddisk_AIX_PPC`, `Harddisk_WINNT_X86` and `Harddisk_LINUX_X86`. In the non-adaptable parts of the mobile code an adaptor class is used instead of `Harddisk`. The adaptor class `IHarddisk_Adaptor` is being generated out of the functionality interface `IHarddisk` by a generator, similar to the CORBA IDL compiler.

Another aspect of adaptation which has not yet been explained is the resolution of the implementation class name from the functionality interface and the environment. This is done by context awareness as explained in the following section.

## 3.4 Context Awareness

If the mobile code moves to a new host and the adaptor classes decide to adapt the implementation classes, the context awareness must inspect the environment and deliver the name of the right implementation class to a given functionality interface.

E.g. the mobile code intends to execute `getPhysicalMemorySize()` of a `IMemory_Adaptor` object. The adaptor class decides to adapt the implementation class and the adaptation contacts the context awareness in order to get the name of an implementation class of the implementation group `IMemory`.

For this decision the context awareness needs following information:

1. the description of the local environment, where the mobile code executes at present, denoted as *environment profile*
2. the descriptions of the suitable environments of all implementation classes belonging to the implementation group, denoted as *implementation profiles*
3. the function for comparing the environment profile with the implementation profiles

For the comparison of the environment profile with the implementation profiles the equality is not sufficient. If an implementation class has a requirement like e.g. at least version 1.0 of the operating system, all operating systems higher than 1.0 and not only exactly version 1.0 work fine for the implementation class. E.g. in the example of the web browser the `Configuration_UX_NETSCAPE` class is suitable for *UNIX* operating systems and not only for e.g. *Linux*. In this case strict equality would not be sufficient.

### 3.4.1 Profiles and Profile Values

The implementation profile must be specified by the application programmer. The environment has several properties, e.g. default browser, operating system, etc. The environment properties are mapped to a set of *profile values*. A profile value has a type, e.g. default browser, which correlates with an environment property and a value, e.g. *Netscape*, which correlates with the current value of an environment property. Every implementation profile consists of a set of profile values.

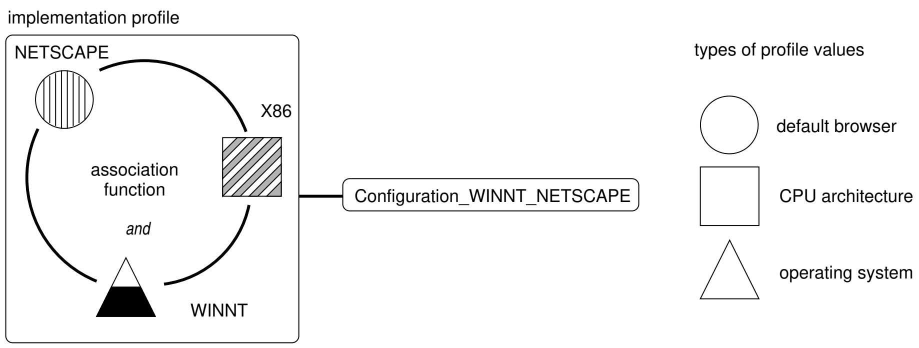


Figure 3.12: Implementation profile

For example the profile of the implementation class `Configuration_WINNT_NETSCAPE` (s. figure 3.12) has profile values of three different types: *operating system*, *CPU architecture* and *default web browser*. The values of the profile values are: *WINNT, x86* and *Netscape*. All three properties must be fulfilled in an environment, that the implementation class `Configuration_WINNT_NETSCAPE` works properly. Therefore the profile values are associated with *and*.

The implementation profile must be compared with the current environment in order to determine whether the implementation class is suitable for the environment. For this comparison each profile value is checked whether it matches the current value of an environment property. The equality may not be sufficient, e.g. for the test of an operating system version, it is sufficient, when a version newer than 3.0 is installed rather than exactly 3.0. A *matching function* is associated with every implementation profile value and describes how this profile value can be matched with the profile value of the environment.

For the matching between a profile value and the value of the environment property a profile value is generated out of the environment property. The generation of profile values out of the environment is specified by the *generating function*. The result of the matching functions is *true* (= profile values match) or *false* (= profile values do not match). The generating function is also contained within the profile value and must be executed in the environment for which the implementation class must be suitable, i.e. the environment, in which the mobile code is running at present.

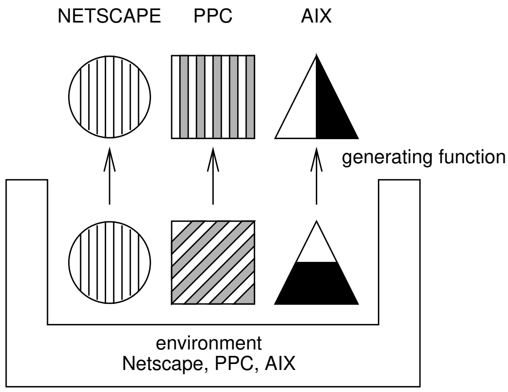


Figure 3.13: Generation of environment profile values

In the case of the implementation class `Configuration_WINNT_NETSCAPE`, the three profile values generate the complementary profile value of the current environment. Assuming the environment is a PowerPC running AIX with Netscape as default browser, the profile values of the implementation class, generate the profile values *NETSCAPE, PPC* and *AIX* (s. figure 3.13).

The association of the results of the matching function is defined by the *association function* as part of a profile. For instance, an implementation class is suitable for a system running *Linux* or *AIX*. The implementation profile has two profile values of the type operating system and they are associated by *or* because the implementation class is suitable for *Linux or AIX*. The result of the association functions determines whether the implementation class is suitable for the environment or not.

The generated profile values are matched against the profile values of the implementation class, i.e. `Configuration_WINNT_NETSCAPE` (s. figure 3.14). The results of the matching function are associated as defined in the association function, i.e. association by *and*. The result of the evaluation of the implementation profile in the example environment is negative. The context awareness must proceed to evaluate other implementation profiles of the same implementation group.

In general the generating function of one type of profile value is the same. In the case of the profile values of the type *default browser* they depend on the environment. For Unix systems, the configuration file in the home directory is checked, but on hosts running Windows, several entries in the Windows Registry must be checked. This principle of context awareness is described in the following section.

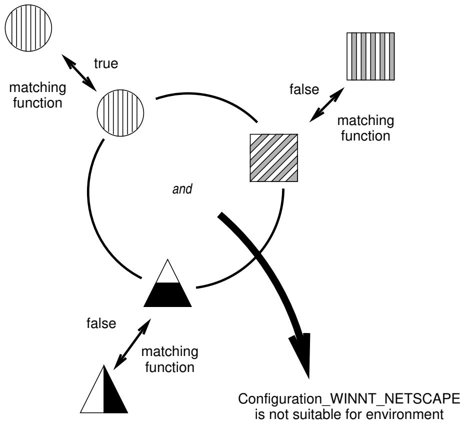


Figure 3.14: Matching of profile values

| functionality interface | implementation class | environment |
|---|---|---|
| `IDefaultWebClient` | `DefaultWebClient_UX` | Unix |
| | `DefaultWebClient_WINNT_X86` | Windows NT, x86 |

Table 3.2: Implementation classes of example

### 3.4.2 Recursive Context Awareness

Another issue of context awareness is the nature of the set of profile values which are used to describe the environment. A fixed set of basic profile values has the advantage, that the functions for acquiring these parameters can be integrated into context awareness efficiently. The disadvantage is that the application programmer is restricted to the set of profile values offered by the context awareness. This suggests that context awareness must be open to allow the application programmer to define new profile values.

E.g. the implementation group `IConfiguration` selects the suitable implementation class, depending on the installed default browser. This is an application dependent type of profile value and cannot be provided by default in context awareness. Another problem concerns the way how profile values are retrieved from the host system. After the mobile code is moved to the new host it can only interact over the standardized common interface of the Java API to figure out more about the environment. In the case of the default browser this API is insufficient. The information about the default browser is saved on Windows operating systems in the registry database which is e.g. accessible via Win32 native C functions, but not through the Java API. On Unix systems it is sufficient to check the home directory for configuration files of a specific browser. Therefore context awareness must not only support the new definition of profile values, but also the adaptation of functions for retrieving system information.

The easiest way to overcome this is by folding the establishment of code for the environment determination into the very adaptation mechanism itself and hence to arrive at a recursive mechanism: *recursive context awareness.* The recursive dependencies must end in axiomatic values which do not need adaptation. In the browser example the information about the default web client must be designed as additional adaptable part of the mobile code.

Table 3.2 shows the implementation classes and their environment profiles which only contain values, *operating system* and *CPU architecture*, that can be determined by Java functions, and resolve the recursive dependency of the implementation group `IConfiguration`. The implementation classes of the implementation group `IConfiguration` need not only to know the operating system and the CPU architecture, but also the default web browser (s. fig. 3.15).

In order to get the information about the default web browser, it is necessary to perform adap-

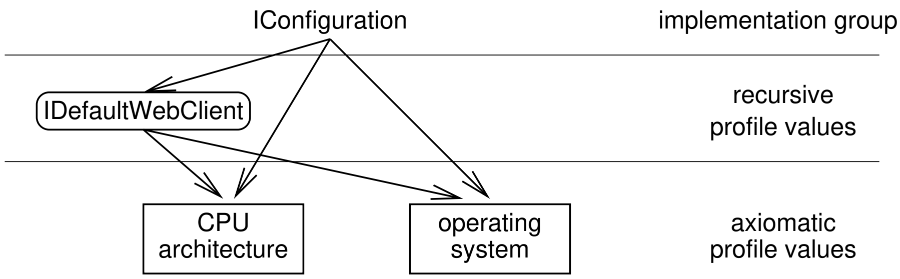


Figure 3.15: Recursive dependencies

tation, because the way to retrieve this information from the system depends on the operating system and the CPU architecture as explained above. This adaptation is realized by the implementation group `IDefaultWebClient`, which is a *recursive profile value*. I.e. a profile value which must perform adaptation in order to execute the generating function. In this case the generating function must create a profile value for the current environment representing the default web browser.

The chain of recursive profile values must end in an *axiomatic profile value*. An axiomatic profile value has a generating function which is environment independent. In the case of the profile value `IDefaultWebClient`, which represents an implementation group and a profile value at the same time, the dependency chain is resolved by the axiomatic profile values *CPU architecture* and *operating system*, which are the only profile values that are needed by `IDefaultWebClient`.

In the cases of *CPU architecture* and *operating system* the environment independent Java function `System.getProperty()` is used to resolve the dependencies.

In the following subsections three different approaches will be presented and compared.

### 3.4.3 Rule Based Context Awareness - Proposal 1

In order to achieve a good level of scalability and flexibility a rule based context awareness can be useful. The idea is to realize the implementation profiles including the association, generating and matching function as rules.

The adaptation obtains the rules as initialization parameter. On every host the rules are interpreted by the adaptation mechanism of the agent and the names of the fitting implementation classes are resolved. For the example of the browser configuration, the rules could look like in figure 3.16.

The section `variables` defines the profile values of the environment which are used and the functions which are executed to retrieve the profile values from the system. It implements the generating function of the profile values. The definition of `predicates` provides the generalization of values, e.g. `unix :=OS in {Linux,AIX}`. This is the realization of the matching function. The association function is defined in the section `matching`. The `matching` section declares the matching function for every implementation class. The implementation classes behind the arrow in the `matching` section are suitable, when the evaluation of their expression returns `true`. The association function is basically realized by logical operators `and`, `or` and `not`.

The application programmer must define the rules for every implementation group. In figure 3.16 the rules for the implementation group *IConfiguration* are shown. If the adaptation wants to load the suitable implementation class specifying the functionality interface, all expressions of the implementation group in the section `matching` are evaluated. The implementation class is selected from the group whose expression returns `true`.

On a host running an UNIX operating system, with Netscape as default browser, the expression of `Configuration_UX_NETSCAPE` will return `true` and the resting expressions of the group return `false`. In this case the context awareness decides that `Configuration_UX_NETSCAPE` is the right implementation class.

```
variables{
OS:=GetOsName;
Arch:=GetArchName;
DefaultWebClient=:dynadap.webclient.IDefaultWebClient.getDefaultWebClient;
}

predicates{
unix:= OS in {Linux,AIX};
nt:=OS in {Windows NT};
x86:=Arch in {x86};
netscape:=DefaultWebClient in {netscape};
iexplorer:=DefaultWebClient in {iexplorer};
lynx:=DefaultWebClient in {lynx};
}

matching{

 ...

 dynadap.webclient.IConfiguration{
   (unix and netscape)
            ->dynadap.webclient.Configuration_UX_NETSCAPE;
   (x86 and nt and netscape)
            ->dynadap.webclient.Configuration_WINNT_NETSCAPE;
   (unix and lynx)
            ->dynadap.webclient.Configuration_UX_LYNX;
   (x86 and nt and iexplorer)
            ->dynadap.webclient.Configuration_WINNT_IEXPLORER;
   ...
 }
}
```

Figure 3.16: Example for context awareness rules

If none of the implementation classes can be chosen, a runtime error is reported by the context awareness to the mobile code, that a class can not be found. If a new operating system, and a suitable implementation class is added to the system, the rules must be extended by a new predicate and an expression for the new implementation class.

The quality of this approach is strongly dependent on the semantic power of the rule language. If it is only a simple language, like in the example shown above, a small number of environments can be described only. The introduction of an additional language – if the rules are considered as a new language – conflicts with the requirement `R1` which does not allow an additional language to the programming language of the mobile code.

The advantage of this technique is the open context awareness which is able to interpret every definition of new profile values or even implementation classes during runtime. If the mobile code does not rely on a static set of rules, but loads the current rules for every adaptation, it is possible to introduce new implementation classes into the system without termination of the core or rewriting of the adaptation mechanism.

If the number of implementation classes increases dramatically, a graphical editor could support the administration of the rule file. The price of these advantages is the loose coupling between the rules and the implementation classes, which is a violation of requirement `R3`, because the rules are totally independent from the mobile code and changes in the implementation require updating the rules in order to avoid inconsistency. Faults can occur very easily, if the consistency between rules and implementation classes is not checked.

Another approach is the introduction of a component model similar to Java Beans [Eng97]. This concept is explained in more detail in the following section.

### 3.4.4 Info-Component-Based Context Awareness - Proposal 2

The *info-component* approach extends the design pattern of functionality interface and implementation classes by *info classes*. To every implementation class an info class is assigned and provides the necessary information of the profile values. The reason for the separation of the implementation classes and the profile values is the basic requirement of dynamic adaptation, i.e. only to load code which is really needed. If implementation classes and the profile values are coupled inseparably, the implementation class must be loaded for context awareness purposes from the repository, but are perhaps never needed for the execution of the mobile code.

An example for the implementation info for `Harddisk_WINNT_X86` might look like in figure 3.17.

```
public class Harddisk_WINNT_X86Info implements ImplementationInfo{
  public static boolean matches(){
    return Environment.os().equals("Windows NT") &
           Environment.arch().equals("x86");
  }
}
```

Figure 3.17: Example of info class

The context awareness loads all info classes and executes their `matches()` method. The context awareness loads the info classes from the repository specifying the functionality interface. The `matches()` method of the info classes implements the matching function and also contains the generating function.

In the example (s. figure 3.17) the generating functions are realized by `Environment.os()` and `Environment.arch()`. For the matching function simple equality is sufficient in this case.

The improvement of this concept against the rules is the higher flexibility of describing and proving the environment. E.g. if the right implementation class from the implementation group `IHarddisk` can be resolved, all implementation infos must be loaded from the repository: `Harddisk_AIX_PPCInfo`, `Harddisk_WINNT_X86Info` and `Harddisk_LINUX_X86Info`.

The name of the implementation class can be resolved out of the class name of the implementation info. The application programmer creates for every implementation class an info class. The name of the info class has as root the implementation class name and the suffix `Info`. E.g. the info class for `Harddisk_WINNT_X86` is `Harddisk_WINNT_X86Info`. This convention must be followed by the programmer, to guarantee that the repository finds the suitable info classes and that the context awareness can resolve the name of the implementation class out of the implementation info class name.

In the info class the environment for the implementation class is defined in the matching function. As shown in figure 3.17, the implementation class `Harddisk_WINNT_X86` works only on a x86 host running Windows NT. In this concept the matching function also contains the generating function, which describes how these profiles values, i.e. x86 and Windows NT, are retrieved from the system. In the example shown in figure 3.17 predefined functions from a class `Environment` are used. For the implementation classes implementing `IConfiguration` a function would be specified that describes how the default browser can be determined.

The disadvantage of this proposal is the loose coupling between the implementation class and the info class. This extension also roughly doubles the number of classes, which must be implemented by the programmer. The handicap of the big number of classes and the loose coupling between implementation classes and their info classes may be minimized by using a graphical programming editor. The graphical programming editor can supervise the coupling between the implementation classes and the info classes. It can enforce the naming convention and support the programmer by a clear presentation of existing implementation classes and info classes.

Nevertheless this workaround can not match the strength of potential structural linking between implementation classes and profile values. A promising approach is presented in the following section, which represents a refinement of the proposed above component model.

### 3.4.5 Integrated Profile Based Context Awareness - Proposal 3

The intention of this concept is the integration of the profiles into the implementation classes but holding down the costs for loading the implementation classes. The implementation classes containing the profiles are loaded into the repository, but only the profiles are transferred to the mobile code over the network.

The loaded implementation classes are instantiated and deliver their implementation profiles to the repository. The context awareness can download the implementation profiles from the repository for resolving the implementation class.

This construction implies that the implementation profiles are coded into the implementation class. The implementation class `Configuration_UX_NETSCAPE` contains the function `getProfile()` which describes the suitable environment for the implementation (s. figure 3.18).

```
public class Configuration_UX_NETSCAPE
            implements IConfiguration,
                       IImplementationClass{
    ...
    public Profile getProfile(){
        return new Profile(new ProfileValue [] {
            new Unix(),
            new Netscape()});
    }
    ...
}
```

Figure 3.18: Example for an integrated profile

The function `getProfile()` is declared by the interface `IImplementationClass` which must be implemented by all implementation classes in addition to the functionality interface, in this case `IConfiguration`. The association function in this concept is limited to the association of profile values by the logical `and` operator. The profile values are objects which contain the following three pieces of information:

1. the profile value itself, e.g. "`Linux`"
2. the commands for getting the profile value from the environment, e.g. `System.getProperty("os.name");`, i.e. the generating function
3. the matching function between the implementation profile value and the environment profile value; i.e. the matching function

Every implementation profile is able to create an environment profile through the generating function and can compare the implementation profile with the created environment profile by applying the matching function.

Figure 3.18 shows how the implementation profile for the implementation class `Configuration_UX_NETSCAPE` is specified within the implementation class. The profile values `Unix` and `Netscape` are classes, which contain both the generating function and the matching function.

In this proposal the profile values are associated with *and*. This association is very strict, because an implementation class may be suitable for mor than one environment, e.g. for *Linux* or *AIX*. But because of mapping profile values to classes a further relation can be assigned to the profile values. The generalization can be constructed by building super- and subclasses. figure 3.19 shows the hierarchy of profile values of the type operating system as the mapped class hierarchy.

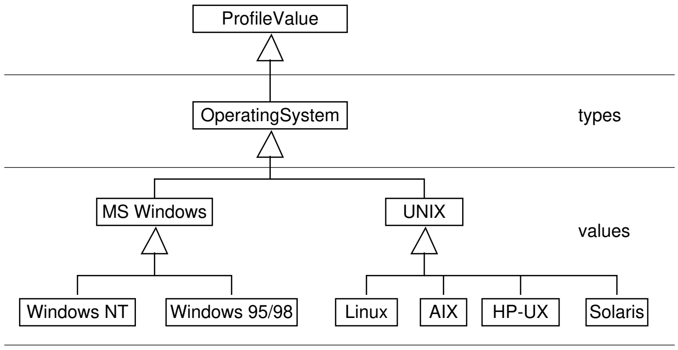


Figure 3.19: Generalization relation between profile values

The matching function between profile values can then be expressed in terms of super- and subclasses:

> *If the profile value of the environment is the same class or a subclass of the profile value of the implementation class, the implementation class is suitable for the environment*

If in the above example an implementation class has the implementation profile value *UNIX*, e.g. `Configuration_UX_NETSCAPE`, it can run on all hosts which can build an environment profile value that is a subclass of *UNIX*, like e.g. *Linux* or *Solaris*.

This is an universal rule, which does not rely on the type of the profile value, and can be implemented by the universal superclass `ProfileValue` for all profile values. The mapping of profile values to classes also offers the possibility to implement more complex expressions than a set of profile values which are evaluated by connecting with `and`.

A set of commonly used types of profile values can be provided as part of the adaptation framework, but application specific types, e.g. default web browser, must be added by the application programmer. A new type of profile class can be created, by extending the `ProfileValue` superclass. In the application specific profile values the generating function must be implemented in order to retrieve the value of an environment property from the system. The same applies to the matching function which matches two values of this type of profile value.

### 3.4.6 Summary of Context Awareness

For the comparison of the presented context awareness concepts three criteria are used:

1. level of integration of profiles into implementation classes
2. programming and maintenance effort when the number of implementation classes increases
3. flexibility in describing environments

The integration classifies how close the implementation profile is tied to the implementation class in terms of coding: e.g. same language, same file. The rule based approach has no relation between the implementation profile and the implementation class. The names of the implementation classes are listed in a different file, even in a different programming language. If e.g. the name of the implementation class changes from `Configuration_UX_NETSCAPE` to `Configuration_WINNT_IEXPLORER` there is no guarantee that the rule file is compulsory updated. The info components are implemented in the same language as the implementation class and they are associated through the common root of their class name. A higher level of integration is achieved by the last proposal, where the implementation profiles are coded in the implementation class itself.

The scalability addresses the applicability of the concept when the number of implementation classes increases. For a higher number of classes the rule based approach still offers an acceptable solution, perhaps supported by an editor with syntax high-lightning or a graphical editor. The info component model has an unacceptable level of scalability. The number of classes doubles and results in higher programming effort, but also in a more complex maintenance. The integrated profiles are not influenced by the number of implementation classes because no further classes or files are needed apart from the implementation classes which are necessary anyway.

The flexibility addresses the way to describe environments. The more flexible the description mechanism is, the more environments and more precisely can be specified. The rule base context awareness relies strongly on the power of the language and is therefore ranked on a middle level of flexibility. High flexibility have both the info components and the integrated profiles. They can both use Java language constructs for the association of profile values and define own matching functions.

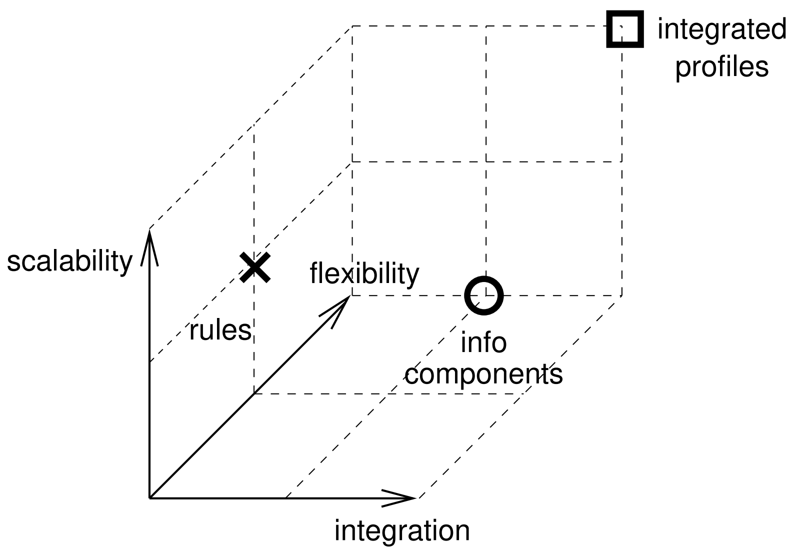


Figure 3.20: Cirteria for the evaluation of context awareness concepts

Figure 3.20 shows a graphical summary of the evaluation of the various context awareness concepts. There are three levels for every category: e.g. no scalability, acceptable scalability, good scalability. From this evaluation the conclusion can be drawn that integrated profiles represent the most suitable approach for context awareness and will be used for the implementation of the prototype.

A very subtle point of this concept is the design of the repository, which is responsible for loading implementation classes and delevering the implementation profiles to the adaptation mechanism.

## 3.5 Repository

The repository provides a service for delevering the adaptable parts to the adaptation reconfiguration mechanism as shown in figure 3.2. For the integrated profile based context awareness the profiles are also loaded from the repository.

The repository allows the mobile code to load only the code over the network which is actually needed for the execution in the current environment. The concept of the repository has to overcome several disadvantages.

If the mobile code is a mobile agent, like in the example of browser configuration, the repository restricts the autonomy of the mobile agent. If a link between the repository and the mobile agent is down, the agent can not proceed with its execution because of missing classes. This contradicts the requirement R6 of chapter 2 because autonomy, a key feature of the *mobile agent* application, is limited by employing adaptation.

Another problem is the increased latency for remotely loading the classes from the repository instead of loading the classes from the local file system.

These implicit disadvantages can not be solved entirely, but their influence can be reduced by extending the framework with *proxy repositories*.

### 3.5.1 Proxy Repository

The proxy repository is proposed to minimize the impact of remote code loading on the runtime behavior of the mobile code. If the connection between proxy repository and mobile code is based on fast and stable links, low latency and high reliability for the procedure of remote loading can be achieved. Hosts which are connected over fast and stable links build a *neighborhood*. The most desirable residence of a proxy repository is in the neighborhood of the mobile code. E.g. a LAN segment builds a neighborhood, but represents only a small number of hosts. This implies a big number of proxy repositories that must be set up if a mobile code moves to a lot of hosts in a complex network. The nature of the neighborhood is strongly application dependent. If e.g. neighborhood is defined as a subnet and a mobile agent visits exactly one host in the subnet, e.g. for configuration of DNS servers, a proxy repository is created on every host that is visited by the agent. In this scenario the concept of the proxy repository would not meet its goal, due to the unsuitable realization of the term neighborhood.

The proxy repositories can be started as a separate thread or process by the adaptation mechanism itself when it can not find a repository in the neighborhood. The started proxy repository serves to the mobile code as a replacement of the *central repository*. The mobile code does not contact the central repository any more, but the proxy repository for loading profiles or implementation classes instead.

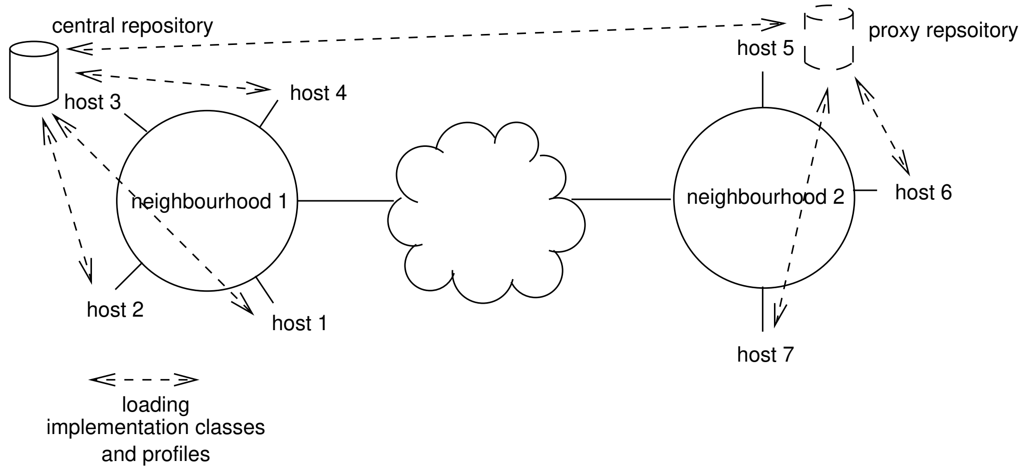


Figure 3.21: Example for proxy repository

Figure 3.21 shows two neighborhoods and the route of a mobile code that moves in turn starting on *host 1* to all others and terminates on *host 7* as the last host. When the mobile code moves from *host 4* to *host 5* the neighborhood, e.g. the subnet, changes and the adaptation mechanism starts a proxy repository on *host 5*. The initiation of the proxy repository and the strategy for loading objects from the central repository to the proxy repository depends on the architecture of the proxy repository, which is covered by the following section.

### 3.5.2 Architecture of the Proxy Repository

Both the implementation of the term neighborhood and the architecture of the proxy, influence the efficiency of the proxy repository.

In this section two different approaches for a repository architecture are presented and one will be selected for the implementation of the prototype.

#### Replication - Proposal 1

Maximal independence from the central repository can be realized by a replication of the central repository. After the creation of the proxy repository all implementation classes and profiles are copied from the central repository to the proxy repository. After this initialization the connection between the central and the proxy repository can be released without any impact on the execution of adaptable and mobile codes.

In figure 3.21 a replicated repository on *host 5* would load all implementation classes from the central repository at the initialization. After the initialization the hosts in *neighborhood 2* do not rely anymore on the central repository.

A replicated repository augments the independence of a mobile agent, but it takes longer to start it, because all implementation classes must be copied from the central repository to the proxy. The start-up time of the proxy repository can be reduced, by an implementation as a cache instead of a replication as proposed in the next section.

#### Cache - Proposal 2

If the proxy repository uses a cache, no implementation classes must be copied at start-up. If the mobile code requests an implementation class it is stored in the proxy repository and other mobile codes in the neighborhood requesting the same implementation classes can be satisfied by the stored classes in the proxy. If implementation classes are requested, which are not already stored, the proxy repository must contact the central repository to obtain these implementation classes.

In figure 3.21 a proxy repository on *host 5* would load the implementation classes from the central repository when they are requested for the first time from the hosts in *neighborhood 2*. The hosts in *neighborhood 2* do not see the central repository and direct all requests to the proxy repository.

The connection to the central repository can only be released by the proxy repository under the assumption that the hosts in the neighborhood offer a homogeneous environment to the mobile code. If this assumption is valid, the same implementation classes are requested by a mobile code on all hosts in the neighborhood. After the mobile code has visited the first host and executed its task, all implementations which are needed for the execution of the task are stored in the cache of the proxy repository and further request can be satisfied by the content of the cache.

## 3.6 Implementation of a Prototype System

After the development of concepts for adaptation in the former sections the implementation of a prototype system will be presented in this section.

The implementation of the adaptation framework is both independent of the configuration task and independent of the agent system. It provides the infrastructure for the remote loading of profiles and implementation classes, the evaluation of profiles and the dynamic transparent linking of implementation classes into the core. This presumes that the mobile code is programmed according to the design pattern as presented in section 3.3.1.

The configuration of the browser relies on the adaptation mechanism. It implements the configuration of a set of web browsers running on various operating systems and CPU architectures. The configuration is brought to the different hosts by a mobile agent using the Voyager agent system platform [Obj00].

Because of the independence of the adaptation framework it is convenient to describe it first and then to continue with the implementation of the remaining parts of the prototype system.

### 3.6.1 Adaptation Framework

#### Architecture

The adaptation includes three aspects as discussed in chapter 3:

1. Reconfiguration
2. Context awareness
3. Repository

The reconfiguration consists of a suitable programming discipline, which is expressed as a design pattern. (s. section 3.3.1, the concept of adaptors, including the adaptor generator, and the loader). The implementation of the reconfiguration is limited to the adaptors and the loader. The design pattern must be followed by the application programmer at compilation time.

Adaptors are generated for the core and are an integrated part of the core. The loader remains encapsulated in the adaptation mechanism. It is steered by the result of the context awareness.

The context awareness delivers the result of the evaluation of the profiles to the loader, which loads the right implementation class over the network from the repository. The profiles are also loaded by the context awareness from the repository. Both the context awareness and the loader rely on the service provided by the repository.

The communication with the repository is integrated in a repository client. The context awareness and the loader send their requests as local method calls to the repository client. The repository client communicates with the repository using a proprietary subset of the *hypertext transfer protocol* (HTTP). The *adaptation*, *context awareness* and the *loader* are designed as Java classes as shown in figure 3.22.

The repository is a separate program that functions as a web server for the context awareness and the loader. Both services make use of the repository client in order to translate between Java method calls and HTTP protocol elements.

The adaptor class `IConfiguration_Adaptor` is generated from the functionality interface `IConfiguration` by the adaptor generator. To the core the `IConfiguration_Adaptor` class offers the two methods `setDiskCache()` and `setMemoryCache()` as defined in the interface `IConfiguration`. In the interior it access the adaptation mechanism.

#### Adaptor

For each functionality interface a different adaptor class is generated. The adaptor class consists of a part that is common for all adaptor classes, denoted as *generic part*, and a part that depends on the functionality interface, the *interface specific part*.

**Structure of an Adaptor** Figure 3.23 shows the adaptor class for the functionality interface `IMemory`. The adaptor class implements the `Serializable` interface which enables the serialization of the adaptor class. This is necessary for moving the adaptor class with the mobile code.

Through the object `m_adaptation` an adaptor can load the right implementation class for the current environment. The member variable `m_adaptation` is marked as `transient`. Though the

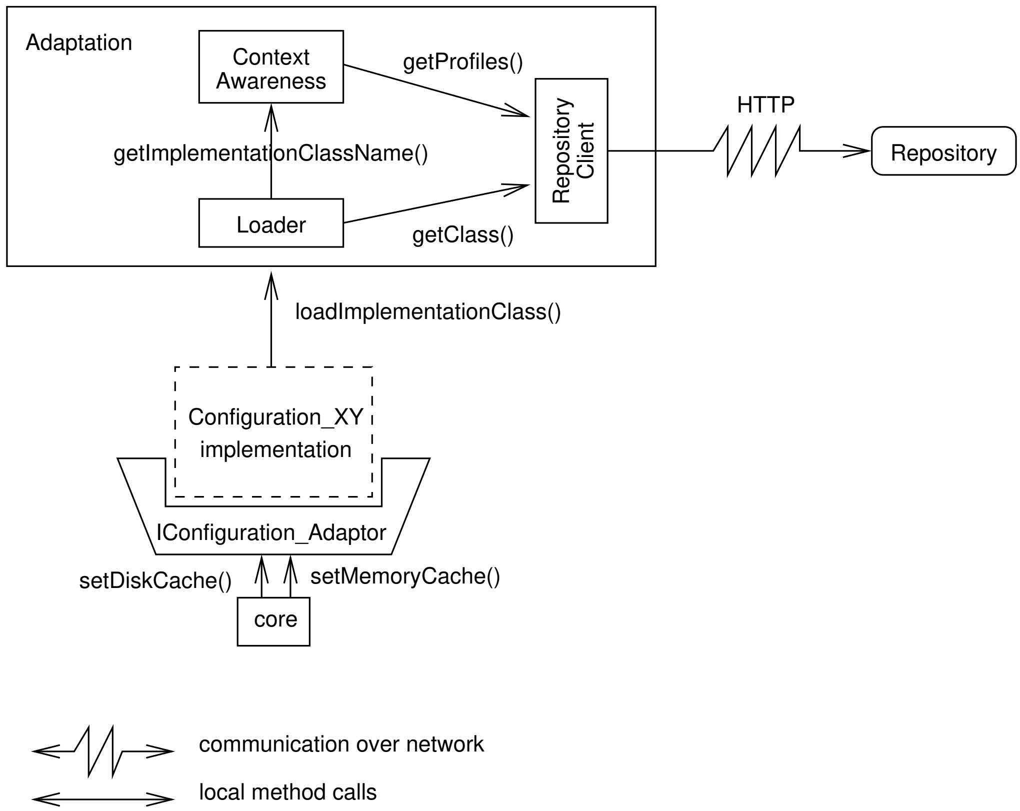


Figure 3.22: Overview of adaptation architecture

adaptor class is serialized for migration to another host, the `m_adaptation` object is dropped and recreated after de-serialization. This reduces the size of code that moves over the network.

The implementation of the interface `IMemory` enables the core to use the adaptor class `IMemory_Adaptor` like a non-adaptable conventional implementation of `IMemory` without any considerations about adaptation.

The member variable `m_iimplementation` holds a reference to an object of the implementation class, which is suitable for the current environment. It is also marked as `transient`, since the implementation class may become obsolete, when the mobile code leaves the current environment.

The generic part of the adaptor classes, printed in boldface in figure 3.24, consists of the implementation of the interface `Serializable`, the member variable `m_adaptation` and the method `createAdaptation()`. The method `createAdaptation()` recreates the dropped adaptation object after the migration to a new environment.

The interface dependent part, printed in boldface in figure 3.25, consists of the method implementations defined in the functionality interface – in this case the implementation of the method `getPhysicalMemory()` – and the object `m_implementation` of implementing the functionality interface.

The adaptor classes provide the skeleton to delegate the method invocations from the core, e.g. `getPhysicalMemory` (s. figure 3.26), to the implementation class and return the result to the core.

Exceptions that are thrown during adaptation are caught by the adaptor class. Two kind of exceptions can be thrown because of adaptation:

- the repository is unreachable for the mobile code
- no profile matches, i.e. there is no suitable implementation class for the environment,

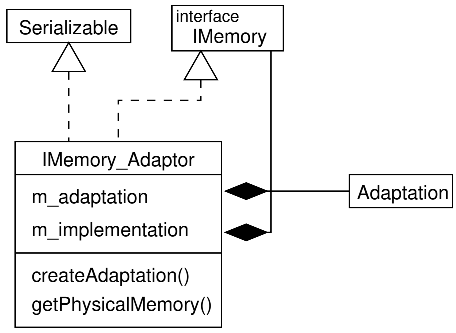


Figure 3.23: Adaptor class for `IMemory`

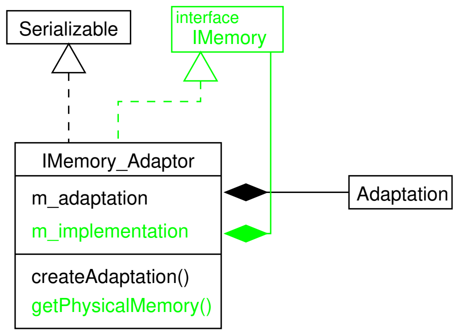


Figure 3.24: Generic part of the adaptor class for `IMemory`

in which the mobile code is running at present

These two kinds of exceptions are both mapped onto a *class not found* exception, which is the consequence for the non-adaptable core: a necessary class can not be loaded.

**Adaptor Generator** The adaptors are generated from a stand alone tool. An application programmer defines the functionality interfaces and the adaptor generator creates the according adaptors. This is done before the application is compiled.

Figure 3.27 shows the different phases of the adaptor generator and the representation of the functionality interface respective the adaptor class. The functionality interface must be available as class file. The generator loads the functionality interface and obtains a representation of the functionality interface as a Java `Class` object.

The structure of the functionality interface can then be inspected by Java Reflection [Fla97]. This allows to pick up information that has already been processed by the normal Java compiler and avoids complicated parsing. In order to create the adaptor class following information is read out of the `Class` object:

- name and package of the functionality interface, e.g. `IMemory`
- method definitions (method name, modifiers, parameters, return value)

The interface name is mapped to `<interface name>_Adaptor`. The method definitions are mapped to method implementations. The method implementations invoke the adaptation mechanism in order to load and instantiate the implementation class, if necessary and delegate the method invocation to the loaded instance of the implementation class.

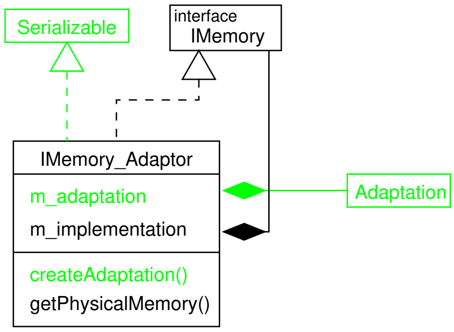


Figure 3.25: Example for interface dependent part of the adaptor class for `IMemory`

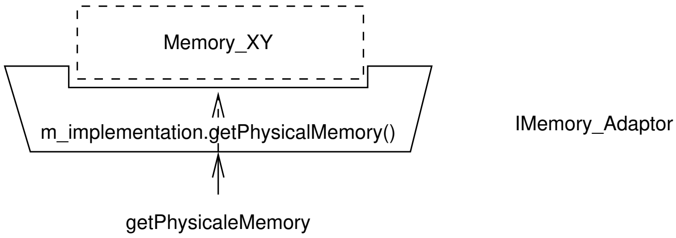


Figure 3.26: Delegation of method call to implementation

In addition to the mapping the generic part is added, which contains the member variable for the instance of the adaptation mechanism and the method `createAdaptation()` in order to instantiate the adaptation mechanism.

### Adaptation

The purpose of the `Adaptation` class is the coordination of context awareness and loader. The adaptor loads an implementation class by specifying the functionality interface of the implementation class, e.g. `IConfiguration`. Before the loader can load an implementation class from the repository the name of the implementation class must be determined by the context awareness.

**ContextAwareness** The class `ContextAwareness` loads the profiles of all implementation classes of the specified functionality interface. The request is sent to the repository client and the context awareness obtains an array of profiles from the repository client.

The functionality of the context awareness is contained in the profiles. The `ContextAwareness` class invokes the matching function of each profile. The implementation class with the unique profile, that executes the matching function successfully is determined as the right implementation class for the current environment. The *profile checker* programm (described in detail below) supports the application programmer to verify that the implementation profiles are unique. If the context awareness detects at runtime that two profiles match the same environment, an exception is thrown. This internal adaptation exception is mapped to a *class not found* exception which is reported to the core.

The name of the implementation class can be retrieved from its profile and is returned to the adaptation.

**Loader** With the information about the right implementation class the loader is able to request the class file from the repository client. The loader is a subclass of the abstract class

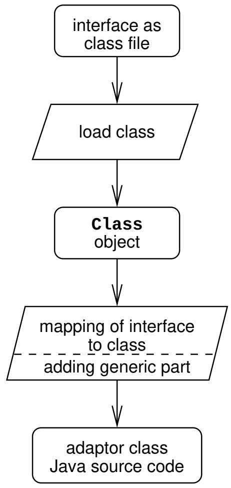


Figure 3.27: Phases of adaptor generator

`java.lang.ClassLoader` contained in the *Java Development Kit* (JDK) [LB98]. The loader implements the `findClass()` and overwrites the `loadClass()` method, which are responsible for locating the class file as raw byte stream and the definition and resolution of a `java.lang.Class` object.

For reducing the communication overhead the default class loading is delegated to the parent class loader, which loads classes from the local file system.

The adaptation loader has access to the list of all implementation classes. If an implementation class is requested by the mobile code the loader contacts the repository via the local repository client.

The loader of the prototype is also able to load dynamic libraries which are necessary for the support of the *Java Native Interface* (JNI) [Lia99]. The JNI supports the execution of native code from Java programs. E.g. this is necessary in order to access the Windows registry, the central configuration database for Windows applications. In the example of the browser configuration the access to the registry is necessary to determine the default web browser and to configure the IExplorer.

The native functions must be compiled and linked into a dynamic shared library, e.g. a DLL for Windows. During the execution of the Java program the library is dynamically linked to the Java program. Before the library is linked to the program the `findLibrary()` method of the loader is invoked by the Java interpreter. It returns the absolute path name of the dynamic library. The loader contacts the repository through the repository client and stores the dynamic library in a temporary directory.

This implementation works fine for the prototype, but the writing of the library to the temporary directory slows down the adaptation procedure.

**RepositoryClient** The repository client transforms the requests of the context awareness and the loader from local method calls to remote HTTP requests. It hides the network communication from the context awareness and the loader and simplifies the protocol between the repository and the adaptation mechanism of the mobile code. A specification of the HTTP requests is desribed below (s. figure 3.34 and 3.35).

### Profiles

The profiles are the essential mechanism of the context awareness. The implementation classes describe their destinated environment in profiles. The profiles are evaluated by the context awareness in the current environment. If a profile matches an environment, its associated implementation class is suitable for the environment.

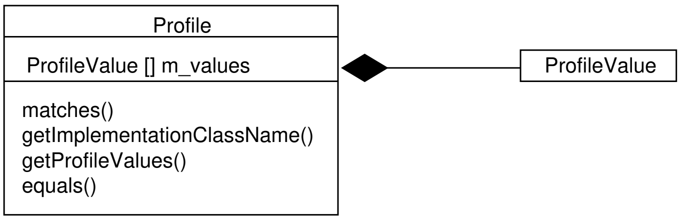


Figure 3.28: Structure of profile class

The profile is implemented as a class called `Profile`. As described the profile contains a set of profile values. In the class `Profile` the member variable `m_values` contains the profile values. The method `matches()` of the class `Profile` invokes the matching function of the profile values and associates the result with *and*. The method `equals` is provided for the profile checker, part of the repository, which helps the application programmer to create unique profiles.

The profile values provide the functionality to match their value with the value of the current environment property, i.e. the matching function. The profile value also contains the functionality in order to retrieve the value of the environment property from the system (generating function).

**Structure of Profile Values** For the implementation of the *Integrated Profile Based Context Awareness* of section 3.4.5, the profile values are mapped to a class hierarchy (s. figure 3.19).

The root of the hierarchy is the abstract class `ProfileValue` which implements only the matching function, in figure 3.19 the matching function is denoted as `matches()`. The generating function is only defined, here denoted as `getEnvProfileValue()`. The other two methods `equals()` and `sameType()` are needed two compare implementation profile values in order two check whether two profiles would match for the same environment.

The various types/properties of profile values are direct subclasses of `ProfileValue`, e.g. the class `OperatingSystem` in the case of the leading example. The types implement the generating functions of the environment property, i.e. `getEnvProfileValue()`. The subclasses of the types represent concrete profile values of the environment properties as separate classes, e.g. the class `Linux`.

The generating functions assumes that the superclasses know all the subclasses. The class `OperatingSystem` must know that there is a class `Linux` that must be built if the name of the operating system is "Linux". Such a construction works fine as long as the set of subclasses is fixed and not intended to be extended. This would constrain the reuse of profile values or requrie re-writing of the profile values if new types and value are added to the profile values hierarchy.

In figure 3.29 the profile value *UNIX* knows the operating systems *Linux*, *AIX* and *Solaris*. The different Unix flavors are implemented as subclasses of the superclass `UNIX`. If a new Unix operating system, e.g. *BSD*, would be added, the generating function of class `UNIX` must be re-written, in order to support the generation of an object of the class `BSD`. Besides of the re-writing of code, the *UNIX* profile value would change its semantic for already existing applications using the class `UNIX` as a representation for Unix flavors.

The solucation is provided by introduction of a new Unix profile value, e.g. *Ext UNIX* (s. figure 3.30). This new profile value is used as re-presentant of Unix in new applications. It knows how to generate a profile value on a host running BSD. If the context awareness runs in another operating system it requests its superclass, i.e. `UNIX`, to generate a profile value.

This approach is not sufficient for the matching function. If an implementation is suitable for *UNIX* operating systems, including *BSD*, e.g. `Configuration_UNIX_NETSCAPE`, its profile contains the profile value *Ext UNIX*. If the mobile code moves to host running *AIX*, the context awareness applies the matching function as defined and implemented in the superclass `ProfileValue`: *If*

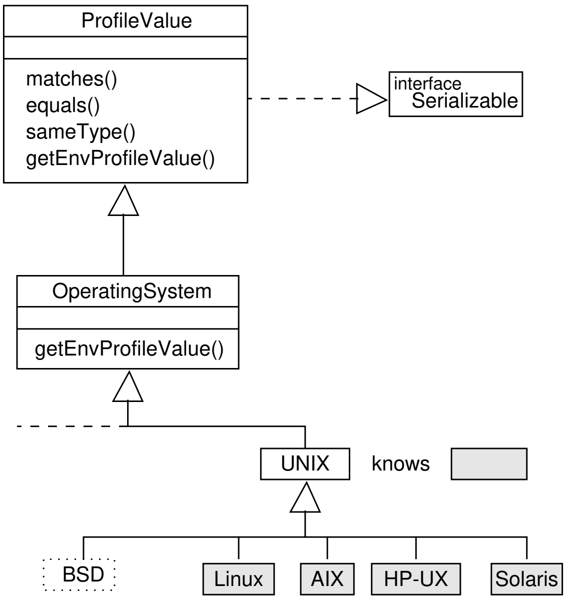


Figure 3.29: Extension by profile value *BSD*

*the profile value of the environment is the same class or a subclass of the profile value of the implementation class, the implementation class is suitable for the environment*

The rule fails, because the class `Linux` is not a subclass of `Ext UNIX`, though the implementation class is intended to work also in a *Linux* environment.

Part of the solution is provided by the *Marker Interface* design pattern [Gra98]:

> "The Marker Interface pattern uses interfaces that declare no methods or variables to indicate semantic attributes of a class."

The interface `IExtension`, which defines no methods, serves as marker interface. The class `Ext UNIX` implements the interface `IExtension`(s. figure 3.31).

The *semantic attribute* of `IExtension` is that classes which implement the interface are not considered to be superclasses in terms of the matching function. Though the class `Ext UNIX` is a superclass of `BSD` in the Java class hierarchy, it is not a superclass in the context of the matching function applied on `BSD`. The class `Ext UNIX` is ignored and `UNIX` is taken as superclass. This relation is shown in figure 3.32.

If *Ext UNIX* is defined by an implementation class it includes both the own sub profile values, i.e. *BSD*, and the profile value under the super profile value *UNIX*

The inspection of the classes, e.g for determining whether a class is marked by `IEtension` is done by using Java Reflection.

**User Defined matching function** As shown in section 3.4.5 the association of profile values by `and` is very strict. This constraint can be released by overwriting the matching function of a profile value of the superclass `ProfileValue`. The evaluation of the default matching function is based on the structure of the profile values. This function can be exchanged when a new profile value is introduced by the application programmer.

The new matching function can compare the implementation profile value and the environment profile value in a completely different way. The result must be of the type `boolean`.

The profile values are encapsulated by the profile, which is defined by the programmer of the implementation class. As suggested in *Integrated Profile Based Context Awareness* in section 3.4.5 the profiles are integrated into the implementation class.

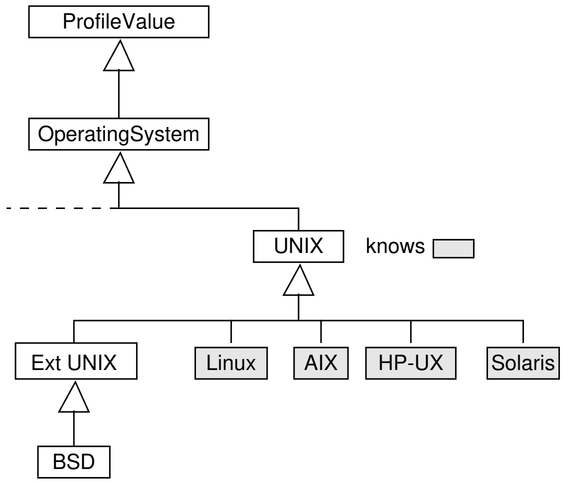


Figure 3.30: Introduction of superclass for extension

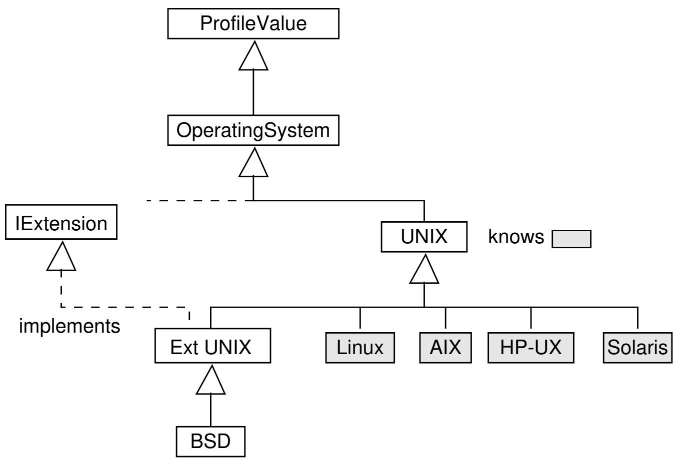


Figure 3.31: Hierarchy of profile value classes in class diagram

**Integration of Profile Values into Implementation Classes** An implementation class is suitable for a certain environment. This environment is described as a profile, which is integrated into the implementation class. The profiles are implemented as a class `Profile` which contains as member variable an array of `ProfileValues`. Every implementation class must implement an interface `IFunctionality` which defines a function `getProfile()`. The return type of `getProfile()` is `Profile`. An application programmer must implement the function `getProfile()` in every implementation class. In the body of `getProfile()` the profile values must be defined in order to create a `Profile` object.

The code example of figure 3.33 is an excerpt from the implementation class `Configuration_UX_NETSCAPE`. `Configuration_UX_NETSCAPE` works properly on a *x86* host running *Windows NT* and with *Netscape* as default browser.

The context awareness service obtains the profiles from the repository, which instantiates the implementation class and invokes the `getProfile()` function. The profiles are serialized and sent to the context awareness. The implementation of the repository is the subject of the following section.

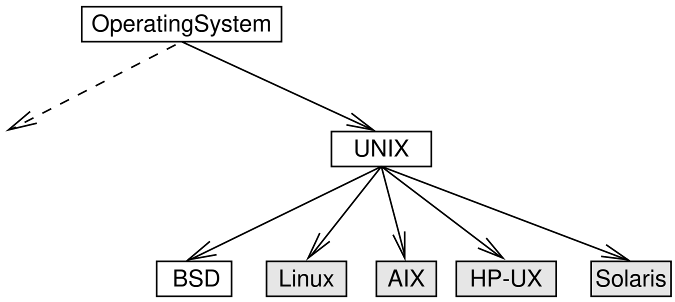


Figure 3.32: Logical hierarchy of profile values due to marker interface

```
public Profile getProfile(){
    Profile result = new Profile(new ProfileValue [] {
        new WindowsNT(),
        new X86(),
        new Netscape()
    });
    return result;
}
```

Figure 3.33: `getProfile()`

**Profile Checker** A second tool of the adaptation mechanism beside the adaptor generator is the *profile checker*. Its task is to verify that there is not more than one implementation class of an implementation group suitable for the same environment. This is tested by analyzing the profiles. The analysis is performed at compilation time to support the application programmer and at the start-up of the repository.

Because of the flexible nature of the profile values it is not possible to obtain a non-ambiguous result of the equality of two profiles. Three cases are possible:

1. Two profiles are equal, i.e. the according implementation classes are suitable for the same environment $\Longrightarrow$ Error
2. Two profiles can not be compared because of complex structure of profile values, no conclusion $\Longrightarrow$ Warning
3. No profiles are equal $\Longrightarrow$ O.k.

For the comparison of two profiles the sets of profile values must fulfill following condition:

*intersection of types must not be empty*

If this is valid it is not guaranteed that it can be decided for all profiles of an implementation group whether they are valid, because if the intersection is a real subset of the set of profile values of one profile and the profile value in the intersection are equal.

The comparison mechanism is implemented in the class `ProfileValue` which is the superclass of all profile values. If a profile value implements a different matching function (s. section 3.4) the `equals()` function must be overwritten, as well.

### Repository

The repository is a stand alone application. It works independently of the adaptation mechanism integrated over adaptors into the mobile code. It provides profiles, implementation classes and if needed libraries to the adaptation mechanism of the mobile code.

The `main()` method of the `Repository` class listens on a defined well-known port for requests. When a request is accepted the repository creates a `Repository` object. which is responsible for answering the request.

There are three types of requests:

1. profile request
2. class request
3. library request

The *profile request* is sent by the context awareness specifying the functionality interface, e.g. `IHarddisk`. The repository returns the profiles of the implementation classes for the implementation group.

The *class request* is initiated by the loader. The loader loads an implementation class by contacting the repository. The loader also contacts the repository if a dynamic library must be loaded, i.e. *library request*.

**Profile Request** For the profile request the repository loads all implementation classes. The set of implementation classes is specified in a configuration file. If a new implementation class is added to the system, the list of implementation classes must be updated. An alternative solution is the bundling of the implementation classes in a special JAR file. It avoids an additional configuration file, but if the set of implementation classes changes, the JAR file must be modified instead of changing a line in a configuration file.

The loaded implementation classes are instantiated on the repository. From the objects of the implementation classes the profiles are retrieved by invoking `getProfile()`. The array of profiles is serialized to a byte stream and sent to the context awareness.

The instantiation of the implementation class on the repository is necessary in order to get the profiles out of the implementation class. In the prototype implementation all instances of implementation classes are held on the repository. If there are a lot of implementation classes, this implementation may lead to high resource consumption. An alternative would be to instantiate the implementation classes for retrieving the profiles once and to drop instances of implementation classes and to keep only the profiles in memory.

**Class Request** The loader specifies the full name of the implementation class including package name and sends it as a request to the repository. The repository looks up in the configured class path for the class file of the implementation class. The class file is converted to a byte stream and sent to the loader.

**Library Request** The library request works similar to the class request. The only difference is the look-up mechanism. The repository looks in a configured directory for the library.

### Communication Protocol

The communication between the repository and the adaptation mechanism is based on a small subset of the HTTP protocol [FGM+99]. It is extended by the definitions for the profile, class and library request.

```
send request message ::= GET /<request>.<request type> HTTP/1.1 CRLF CRLF
request ::= <functionality interface> | <class name> | <library name>
request type ::= interface | class |  library
```

Figure 3.34: Format of *send request* message

Two messages are supported by the protocol:

1. send request

2. send answer

The *send request* message format is structured as shown in figure 3.34. The *send answer* message format is structured as shown in figure 3.35.

```
send answer ::= HTTP/1.1 <status code> CRLF
                Content-Length: <message length> CRLF CRLF
                <message body>
```

Figure 3.35: Format of *send answer* message

As discussed in section 3.5 the central repository has a major impact on the characteristics of the mobile code, e.g. autonomy of the mobile agent. The proposed solution for the prototype is a proxy repository.

**Repository Proxy** The proxy is a replacement for the repository in the neighborhood of the mobile code (s. section 3.5.1). The class `RepositoryProxy` implements a server loop similar to the server loop in the `main()` method of the `Repository` class listening for profile, class and library requests. The requests are satisfied with a cache, i.e. if the requested profiles, classes or libraries are cacheed, the contents of the cache is returned.

In the prototype implementation the proxies and the central repository can be inconsistent, if the implementation classes on the central repository are modified. In order to avoid inconsistencies the central repository could notify all proxy repositories. Therefore, it would be necessary that the central repository knows the addresses of existing proxies.

In an alternative approach the proxies could check the version of the implementation class by contacting the central repository before they deliver an implementation class to the mobile code. This approach would constrain the autonomy of the proxy repository.

A more suitable proposal could be the deployment of a limited lifetime for proxies. If a proxy does not receive a request for a certain period of time it dies. The next time the mobile code moves into the neighborhood of the dead proxy, then a new proxy is created with the current versions of implementation classes. This would also facilitate the management of the proxies, which are distributed over the network and are not needed any more.

**Repository Info** The information about the address of the available repositories must be managed by the mobile code. The adaptation framework provides a class `Info`, which saves the information about available repositories. An object of `Info` is initialized by the central repository and is moved along with the mobile code. After the arrival on a host the `Info` objects check whether the repository which was used on the last host, or any other known repository is in the neighborhood of the current host. If no repository can be found in the current neighborhood the `Info` objects starts a separate repository proxy thread. The address of the current usable repository is published by the `Info` object as Java properties. The Java properties can be read by the adaptation mechanism of the mobile code.

For the prototype the neighborhood (s. section 3.5.1) is defined in a configuration file. A list of hosts separated by line-feed forms a neighborhood. A new list is separated from the former by an empty line. This configurable definition of neighborhoods simplifies the implementation of the prototype but also offers a flexible way to configure different scenarios independently of the actual network configuration.

The repository is the one edge of the adaptation framework in opposite to the adaptors which are strongly integrated into the mobile code.

### 3.6.2 Implementation of Browser Configuration

This section describes the implementation of the sample application. The configuration of the browser consists of two parts. The first part acquires the two system parameters: physical memory and the free disk space. The second part configures the browser according to the parameters speicifed by the user, e.g. 5% of the physical memory, and to the actually available resources, e.g. 95MB.

Both parts can benefit from adaptation in a heterogenous environment, as described in section 3.1. For the configuration of the default browser recursive context awareness is necessary (s. section 3.4.2).

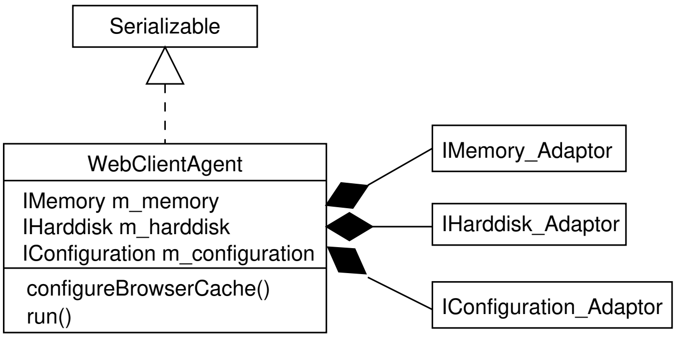


Figure 3.36: Structure of `WebClientAgent`

The browser configuration is executed by a mobile agent implemented as a class holding references to the adaptor classes providing adaptaptable class for acquiring system information (`IMemory_Adaptor`,`IHarddisk_Adaptor`) and setting the browser cache parameters (`IConfiguration_Adaptor`) (s. figure 3.36). The method `configureBrowserCache()` executes the configuration of the web browser using the adaptor classes. The method `run()` is needed for the mobile agent functionality of the class `WebClientAgnet` described in the following sections.

### 3.6.3 Mobile Agent on Voyager Agent System Platform

The configuration of the browser on a host is carried out by a mobile agent, which moves to several hosts, where the configuration task is executed. The mobile agent of the prototype implementation executes the following steps on every host:

1. initialization
2. execution of task
3. computation of the remaining route
4. migration to the next host

For the implementation of the mobile agent the Voyager agent system platform is used [Obj00]. The heart of the Voyager agent system platform is the class `Agent`, which can convert any serializable object into an object of the type `IAgent`. An object of type `IAgent` offers the method `moveTo()`, which realizes the movement of an object to a remote host. `moveTo()` takes as parameters the destination address of the remote host and optionally the specification of a method provided by the moving object. After the movement of the object all local references to the object can be released, though the object is not deleted on the remote host. It can act autonomously.

For the sample application an instance of the class `WebClientAgent` is converted to an object of the type `IAgent`. The method `moveTo()` is called, specifying the destination address of the first host and the method `run()` as the starting point of the execution on the remote host. The method `run()` triggers the browser configuration and invokes after the execution of the task the method `moveTo()` again specifying the next host and itself as execution point (s. figure 3.37).

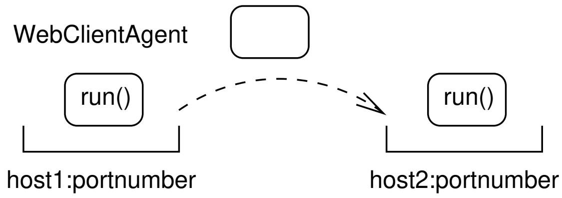


Figure 3.37: Movement of `WebClientAgent` object

The movement and the execution of the configuration of the web browsers continues until all hosts listed for configuration are visited. Then the object dies.

The condition for moving an object to a remote host when using Voyager is a running Voyager server on the remote host. A Voyager server is a separate Java program which listens for requests on a specified port and provides the runtime environment for the received object. The specified port builds together with the host name of the remote host the destination address. The destination address is used as a parameter for the method `moveTo()`.

**Integration of Technologies**

The sample application relies mostly on Java technology, on dynamic adaptation and on code mobility. The adaptation is realized by the presented concept, the code mobility is implemented by Voyager.

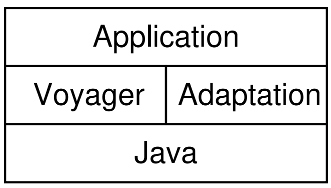


Figure 3.38: Stack of technologies

Figure 3.38 shows how the three technologies are integrated. The Java technology serves as the basis for dynamic adaptation and for Voyager. The Java JDK classes must be installed on every host where an application deploying dynamic adaptation or Voyager is executed.

Since the sample application relies on code mobility based on the Voyger technology the Voyager packages must be installed on every destination host.

The classes needed for adaptation can be divided into classes belonging to the adaptation framework and application specific classes. The framework classes must be installed on every host. The application specific classes are further divided into non-adaptable classes and adaptable classes, i.e. implementation classes.

For the sample application the non-adaptable and environment independent classes must be available on every host. This may be improved by the agent system, in the case of the sample application by Voyager, which allows a distribution of class files by the Voyager agent platform without installation of necessary class files on every destination host.

The installation of the implementation classes is done by the adaptation framework. The classes are loaded if needed from the repository.

# Chapter 4

# Analysis and Evaluation of Methodology and Implementation

This chapter presents an evaluation of the proposed methodology to review whether the requirements set up in chapter 2 are fulfilled and whether the proposed mechanism is useful for deployment under the right circumstances. The basis for the evaluation is the implementation of the prototype, which is e.g. needed to measure the consumption of bandwidth, which is expected to be lower when using dynamic adaptation instead of a conventional monolithic approach. For this purpose an equivalent application has been developed using static customization as described in section 1.2. The following section discusses the fulfillment of requirements `R1-R10` (s. table 2.1). These requirements concern the programming effort and the interaction between adaptation and application. A separate section is dedicated to the revision of requirements `R11-R13`. They concern the runtime overhead when using adaptation. In order to verify these requirements a more complex configuration for measurements has been established.

## 4.1 Meeting the requirements

### 4.1.1 Programming Effort

The prototype is written in Java. For the integration of the adaptable classes into the core no further meta languages are needed. The description of the implementation profiles and the environment profiles is realized in Java. Therefore `R1` is fulfilled up to a certain degree.

Some key concepts are based on reflection. Thus, the concept is only applicable to OO programming languages supporting reflection. Inconvenient in terms of `R1` is the interaction with the framework, which does not support other programming languages than the one used for its implementation. If the adaptation concept is used for an application written in a different programming language, the framework must be ported for the new programming language and cannot be reused.

The second requirement concerning the programming effort considers the overhead in terms of lines of code or number of classes (`R2`). The solution as implemented in the prototype requires programming a functionality interface, which is environment independent, and the development of the implementation classes, one class for each environment.

If it is assumed, that for static customization a separate class for each environment has to be designed, because of modular programming, the number of classes needed for dynamic adaptation is not higher than for static customization.

The explicit description of the environment in every implementation class instead of conditional branches as assumed in static customization, affords more lines of code, but improves the maintainability at the same time.

Requirement `R2` is met it terms of number of classes. The lines of code needed for dynamic adaptation are increased, but it also offers the advantage of better maintainability.

Requirement R3 concerns the placement of the profiles in relation to the implementation class. A strong coupling of profile and implementation classes simplifies the maintenance and avoids inconsistencies. As described in section 3.4.5 according to the proposed methodology the profiles are specified within a method body of the implementation class. This offers a high level of integration and therefore meets R3

For avoiding a broad spectrum of errors during runtime due to adaptation, R4 requires to detect possible failures during compilation. Since the proposed methodology implements the profiles in a conventional programming language, (s. discussion about R1 above), it is guaranteed that the profiles are implemented correctly concerning the syntax. The profile checker, which is integrated into the repository proves the profiles of an implementation class and delivers information about conflicting profiles. Whether the right profile is specified in the implementation, i.e. the profile describes the right environment, for which the implementation class is suitable, is a potential source of errors, which can be detected at runtime only. Other failures might occur due to unreachable repositories or an environment which is not supported by an implementation class.

Such errors must be handled by the application during runtime and are part of the second set of requirements, which covers the integration of an adaptation mechanism into the application. Whether the proposed methodology fulfills these requirements is discussed in the following section.

### 4.1.2 Interaction between Application and Adaptation Mechanism

The definition of the term *dynamic* adaptation, is also stated in R5: adaptation during runtime, i.e. without termination of the application. Since the environment is assumed to be static during the runtime of an application on a host, adaptation is only necessary after moving to a new environment. Thus, the events of migration and adaptation are combined in the proposed methodology. Because most architectures of mobile code, e.g. [Obj00], support weak mobility (s. section 1.1.1) only, there are no executing methods, including adaptable methods. Together with dynamic linking, which is supported by Java, the application needs not to be terminated for adaptation, because the new implementation classes are linked dynamically and the method calls are redirected through the adaptors (s. section 3.3.3).

The semantic of class loading is changed for implementation classes concerning the source of the class files. Instead of being loaded from the local file system or the common platform for the mobile code, the implementation classes are loaded from the repository. This implies that the reliability of the repository and especially of the links between the application and the repository have an influence on the runtime behavior of the application. This problem is formulated by R6, which requires minimal influence on the application characteristics. This requirement is partially fulfilled by the proposed methodology through the concept of the proxy repository. The proxy repository minimizes the negative impact of a central repository.

Other kinds of transparency are required by R7 and R8. The initiation of adaptation and the necessary procedure of linking must be transparent to the application. This is provided by the adaptors. The adaptors are responsible for loading the suitable implementation classes and for the linking, for hiding everything to the application, and for simulating an ordinary Java class, which implements functionality for one environment only. Though the adaptors hide the initiation and the linking to the application, the introduction of adaptors into the application prevents a total transparency. The concept of adaptors introduces a high level of transparency but does not provide total transparency offered by *virtual classes* (proposal 3 in chapter 3).

The concept of adaptor classes offers adaptable methods in standard Java syntax as required by R9. The semantic transparency is postulated by R10, but depends on the availability of a suitable implementation class for the current environment and the reliability of the repository.

## 4.2 Runtime Overhead

One of the advantages of dynamic adaptation is the efficient use of bandwidth. This is emphasized by the requirement R11. The gain in bandwidth is paid by higher runtime. The deployment of adaptation implicates an augmented runtime. The detection of the environment and the dynamic linking of parts into the application affords additional time. The purpose of the requirements R12-R13 is not to postulate an adaptation concept with necessarily lower runtime than when deploying static customization, but to keep at least the overhead of adaptation as low as possible.

In order to approximate the runtime overhead of adaptation, some rough measurements are done comparing the runtime of the prototype and an equivalent monolithic application. This is the object of the following section.

### Setup of Measurements

The goal of the measurements is the comparison of the prototype implementation using dynamic adaptation, called prototype version in the following, and an implementation using customization denoted as customized version.

**Reference Customized Application** The concept of the customized version is based on modified adaptor classes. Instead of containing the class `Adaptation` a adaptor holds references to all implementation classes of the implementation group. The method call for the implementation class is delegated to the right implementation class by traversing several `if-then-else` statements.

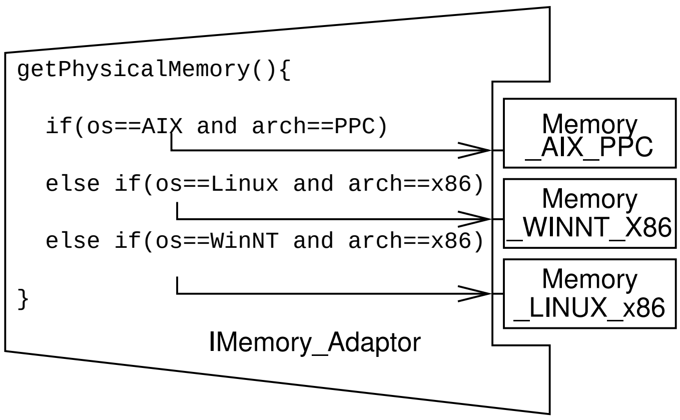


Figure 4.1: IConfiguration adaptor of the customized version

Figure 4.1 shows an example for a modified adaptor for the customized version. The class `IMemory_Adaptor` holds references to objects of all implementation classes, which implement the functionality interface `IMemory`. The method `getPhysicalMemory` is delegated to the right implementation class. The selection of the implementation class is hard coded in conditional branches. This scheme is applied to the other adaptors of the example, i.e. `IHarddisk_Adaptor`, `IDefaultWebClient_Adaptor` and `IConfiguration_Adaptor`. This modification of the prototype is used as the reference application for static customization, which is equivalent to the prototype as implemented according to the proposed methodology.

This design of the reference customized application is chosen, because it allows to reuse most parts of the adaptable version and keeps the effort for implementing a reference application low. A customized version developed from scratch would have a lower runtime, but it is considered to be sufficient in order to approximate the ratio between the runtime of an adaptable application and a customized version.

**Runtime Measurements** The measurements are made by setting a time stamp $t_{before}$ before the method calls of adaptor methods and afterwards $t_{after}$. The difference between $t_{before}$ and $t_{after}$ is taken for the runtime of the method. For the time stamps the global time is taken. In order to get reproducable results, the measurements are done on a single-user host, where the conditions for the measurements can be hold stable for the period of measuring and restored as needed.

Time stamps within the adaptation mechanisms measure the time needed for context awareness and remote class loading for a more detailed analysis of the method runtime. The implementation of the customized version contains only time stamps for the method calls of the adaptor classes. The time which is needed for migration is not measured because the effort for setting up synchronized clocks between the hosts is much higher than the gain of exact results compared to a more simple approximation. The lowest time which is needed for migration is the size of executable code divided by the theoretical bandwidth.

### Results of Measurements

According to the requirement `R11` the size of executable code which moves over the network should be lower when using adaptation, than when using static customization. An objective of this section is to derive an equation from the measured code sizes. The equation should help to decide whether adaptation induces a gain of bandwidth.

The customized application is based on adaptors identical to those of the prototype version using dynamic adaptation. Thus, it is sufficient for the comparison of the size of executable code to compare the size of used implementation classes, eventual dynamic libraries and profiles neglecting the size of the core. The size of the core can be neglected because an identical core is also used for the customized version.

| implementation group | size of serialized profiles [byte] | environment | size of implementation class [byte] | size of dynamic library [byte] |
|---|---|---|---|---|
| IMemory | 1046 | AIX, PPC | 1724 | |
| | | Linux, X86 | 1788 | |
| | | WindowsNT, X86 | 978 | 18209 |
| IHarddisk | 1052 | AIX, PPC | 2415 | |
| | | Linux, X86 | 2599 | |
| | | WindowsNT, X86 | 1090 | 17965 |
| IDefaultWebClient | 891 | Unix | 1275 | |
| | | WindowsNT, X86 | 1369 | 19785 |
| IConfiguration | 1408 | Unix, Netscape | 2607 | |
| | | Unix, Lynx | 2335 | |
| | | WindowsNT, Netscape | 2781 | 20187 |
| | | WindowsNT, IExplorer | 1362 | 19788 |

Table 4.1: Size of executable code

The customized version holds references to all implementation classes over the whole lifetime of the application. When the mobile code moves to a new host, the serialized objects are also moved along. The approximation for the code size of the customized version (s. section 1.2) is the sum of the sizes of all implementation class files of the implementation groups `IMemory`, `IHarddisk`,`IDefaultWebClient`,`IConfiguration` (s. table 4.1):

$$\begin{aligned}\text{size of executable code} &= \sum_{i=1}^{k} sizeOf(\texttt{IMemory}_i) + \sum_{i=1}^{l} sizeOf(\texttt{IHarddisk}_i) \\ &\quad + \sum_{i=1}^{m} sizeOf(\texttt{IDefaultWebClient}_i) + \sum_{i=1}^{n} sizeOf(\texttt{IConfiguration}_i) \\ &= 4490 + 6104 + 2644 + 9085\end{aligned}$$

$$= \quad 22323 \quad [\text{byte}]$$

with

| | |
|---|---|
| $k$ | number of environments supported by implementation group *IMemory* |
| $l$ | number of environments supported by implementation group *IHarddisk* |
| $m$ | number of environments supported by implementation group *IDefaultWebClient* |
| $n$ | number of environments supported by implementation group *IConfiguration* |

The prototype version only holds references to implementation classes which are suitable for the current environment, i.e. one implementation class out of each implementation group. If the mobile code moves to a new host, the objects are dropped. For loading a required implementation class, the context awareness loads the instantiated and serialized profiles of all implementation classes of an implementation group.

The approximation for the code size of the prototype version is the sum of all serialized profiles and the arithmetic mean of the implementation classes. The arithmetic mean of the implementation classes represents the implementation class that is suitable for the current environment and actually loaded. Due to differing sizes of implementation classes for different environments, the arithmetic mean is taken in order to get an environment independent unit of measurement.

$$\begin{aligned}
\text{size of executable code} \quad &= \sum_{i=0}^{k+m+n+l} sizeOf(profile_i) \\
&+ \sum_{i=1}^{k} sizeOf(\texttt{IMemory}_i)/k + \sum_{i=1}^{l} sizeOf(\texttt{IHarddisk}_i)/l \\
&+ \sum_{i=1}^{m} sizeOf(\texttt{IDefaultWebClient}_i)/m \\
&+ \sum_{i=1}^{n} sizeOf(\texttt{IConfiguration}_i)/n \\
&= 4397 + 1496 + 2034 + 1322 + 2271 \\
&= 11520 \quad [\text{byte}]
\end{aligned}$$

The approximations of the size of executable code show that the prototype version ($11520[byte]$) is smaller than the customized version ($22323[byte]$). From this approximations a general equation for `R11` can be derived:

$$\sum_{i=0}^{k+m+n+l} sizeOf(profile_i) + \sum_{j=1}^{k+m+n+l} sizeOf(implementation\ class_i)/(k+m+n+l) \quad < \quad \sum_{i=0}^{k+m+n+l} sizeOf(implementation\ class_i)$$

If the number of profiles and implementation classes increases and the average size of the implementation class is assumed to be constant, the size of profiles and the sum of implementation classes becomes much bigger than the average size of the implementation class. Thus the average size of the implementation class can be neglected and the equation can be established for each implementation class and its profile:

$$sizeOf(profile) \quad < \quad sizeOf(implementation\ class)$$

As long as the size of the profile is smaller than the implementation class requirement R11 is fulfilled.

This equation can be taken to decide whether bandwidth can be saved when using adaptation. This equation is a very rough rule and can only be taken for a first approximation. Especially the sizes of the executable code may strongly depend on the environment. As shown in table 4.1 some implementation classes are rather small, but need to load big dynamic libraries, containing native code. This aspect is not taken into account for the derivation of the equation, because they are only specific for a few environments in this scenario.

A moderate increase of runtime as cost for the gain of bandwidth is the target of R12 and R13. R12 postulates low reconfiguration time and R13 low time for the call of adaptable methods. In the proposed methodology (s. section 3.3) the method call is used as the event for adaptation. Because of this integration of method calls and adaptation the requirements R12 and R13 are jointly evaluated: *low runtime overhead of adaptable methods*. The overhead consists of two parts: the execution of context awareness and the loading of the implementation classes (s. figure 4.2). The adaptation is done, when the reference to a class is accessed in a new environment for the first time. Thus there is only an increased latency for the first method.

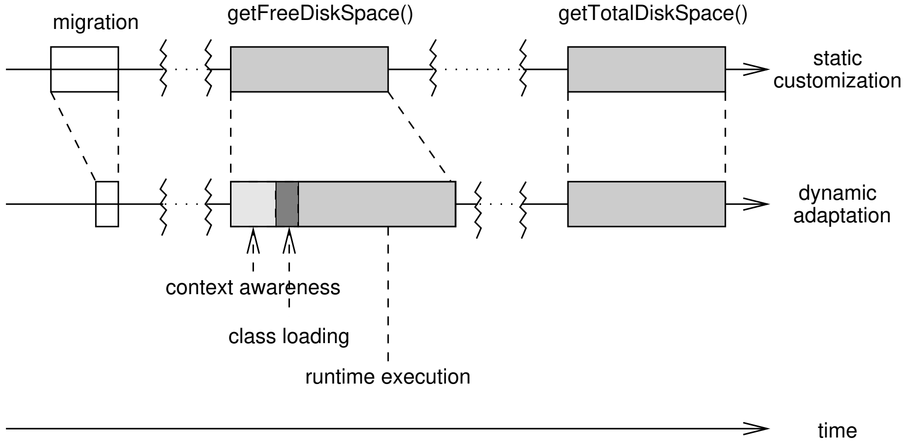


Figure 4.2: Runtime latency due to dynamic adaptation

In the prototype and the customized version the methods of the adaptable classes are called as shown in figure 4.3. The gray colored method calls must carry out adaptation before they can execute their tasks.

The latency due to adaptation in the first method call of the object and the unchanged runtime for succeeding methods is proved by the measurements (s. table 4.2).

| reference | method | average function runtime [ms] | |
|---|---|---|---|
| | | dynamic adaptation | static customizing |
| m_memory | getPhysicalMemorySize() | 346 | 9 |
| m_harddisk | getFreeDiskSpace() | 211 | 83 |
| | getTotalDiskSpace() | 48 | 45 |
| m_configuration | setDiskSpace() | 467 | 20 |
| | setMemoryCache() | 13 | 13 |

Table 4.2: Measured runtime values for methods using dynamic adaptation and static customization

The method getFreeDiskSpace() of the reference m_harddisk has a higher runtime for the prototype version than for the customized version. The method getTotalDiskSpace() of the

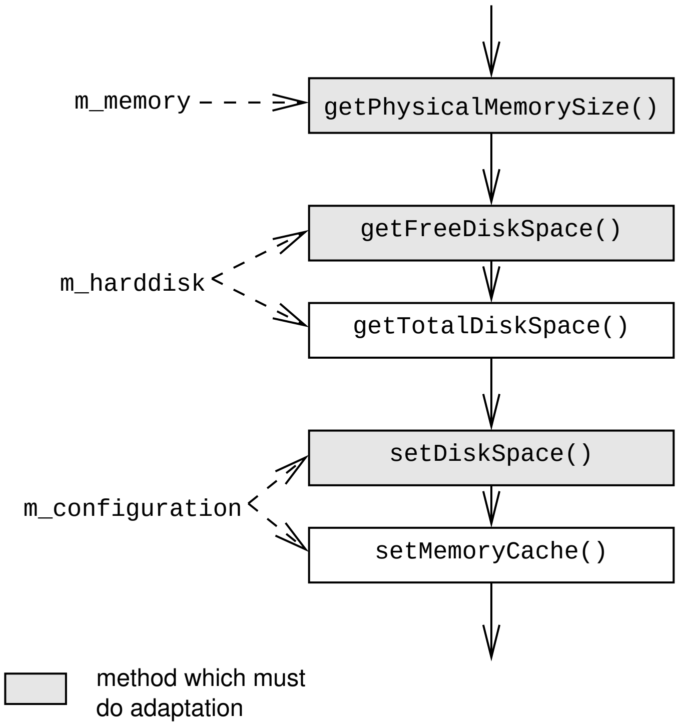


Figure 4.3: Sequence of method calls

reference `m_harddisk`, which is invoked in the program flow after `getFreeDiskSpace()`, has almost the same runtime as the method of the customized version.

The values in table 4.2 are measured on a single-user host with a local repository. Each value in the table is the arithmetic mean of a set of 100 measured runtime values.

| reference | method | dyn. adaptation - stat. custom. | ratio per method |
|---|---|---|---|
| `m_memory` | `getPhysicalMemorySize()` | 337 | 0.97 |
| `m_harddisk` | `getFreeDiskSpace()` | 128 | 0.61 |
| | `getTotalDiskSpace()` | 3 | 0.06 |
| `m_configuration` | `setDiskSpace()` | 447 | 0.96 |
| | `setMemoryCache()` | 0 | 0 |

Table 4.3: Comparison of runtime values

The latency because of adaptation is partially up to 97% (`getPhysicalMemorySize()`) of the total runtime of the method. This is a high ratio and contradicts requirements `R12-R13`. The reason for the high ratio is the short runtime of the method and the delay caused by the adaptation process at the first method call. If the latency of adaptation is considered over all methods of a reference the ratio of latency per method is more acceptable. If the latency for adaptation is assumed to be constant, adaptation becomes cheaper in terms of runtime. The assumption of a constant runtime of adaptation relies on the fact that the biggest portion of runtime is spent for loading the profiles and executing the context awareness. If the complexity of profiles remains at a constant level, the runtime for context awareness and thus for adaptation can be assumed to be constant.

## 4.3 Summary

The proposed methodology and the implementation of the prototype meet most of the requirements. Because of using reflection the concept relies on programming languages which support this technology and the modification of the framework is necessary if another programming language than Java is used.

Complete transparency to the application is however not been achieved. Adaptors hide most of the adaptation mechanism, but are still visible to the application.

The gain of bandwidth depends heavily on the size of the implementation classes and the number of environments. If only small implementation classes for few environments are used, the adaptation version is less efficient than a customized version. The efficency of adaptation increases with the size of implementation classes and the number of environments.

| scenario | customized version | dynamic adaptation |
|---|---|---|
| long running methods | + | ++ |
| big implementation classes | - | + |
| few different environments | + | - |

Table 4.4: Comparison between customized version and dynamic adaptation

The costs for gained bandwidth is a runtime overhead. This runtime overhead becomes negligible, if an application has a long running time on a host.

Table 4.4 gives an overview in which cases adaptation may be a better choice than the conventional customized version.

# Chapter 5

# Conclusions

## 5.1 Contribution

### 5.1.1 Requirements

In order to survey the state-of-the-art of adaptation, a list of requirements for dynamic adaptation of mobile code has been set up in chapter 2. The evaluation of the known technologies concludes that a new methodology must be developed. The requirements serve as guideline for the development of the methodology throughout chapter 3 and help reviewing the resulting concept and its implementation in chapter 5.

### 5.1.2 Design pattern

One part of the proposed methodology as discussed in chapter 3 is a guideline for the programmer for creating adaptable mobile code. The design pattern is based on abstraction by interfaces. The environment independent functionality is defined in the *functionality interface*. The environment dependent implementations are mapped into *implementation classes* which implement the common *functionality interface*.

As evaluated in section 4.1.1 the programming effort for adaptable mobile code compared to static customization is not necessarily higher. The noticeable difference for the programmer is the deployment of the adaptor generator which does not increases the number of classes, but the compilation procedure must be modified. The adaptor generator is part of the adaptation framework.

#### Profile

For the specification of the suitable environment the concept of a *profile*, which is associated with the implementation, is introduced in this work (s. section 3.4.1). The profiles are designated to be evaluated by the framework in order to determine the right implementation. The profile concept is based on object-oriented mechanisms, e.g. relations among implementation requirements and environment properties are mapped to a class hierarchy.

### 5.1.3 Framework

The adaptation framework forms the second part of the proposed methodology. It integrates the adaptation mechanisms as transparent as possible into the mobile code, supports the selection of the right implementation for the current environment and the dynamic linking of the selected implementation.

### Adaptors

The concept for the integration of adaptation into the mobile code relies on *adaptors* (s. section 3.3.3). The adaptors are generated by an adaptor generator and are used within the non-adaptable part of the mobile code as transparent interface to the adaptation mechanism. The generation of the adaptors is necessary in order to provide a code base for the compiler (s. discussion in 3.3.3) but the automatic generation does not lead to a high overhead of programming effort as a consequence.

### Context Awareness

The context awareness as part of the framework evaluates the implementation profiles in the local environment. The implementation profiles are loaded from the repository. It is hidden by the adaptors and contributes its result to the loader.

### Loader

The loading of the right implementation classes is done by the loader. In order to support the execution of native code in the Java environment the loader must support loading of dynamic shared libraries as required by the JNI mechanism [Lia99].

### Repository

The repository provides the profiles, implementation classes and native libraries to the context awareness and the loader. The repository is needed for every adaptation procedure. If the repository is not reachable the adaptation fails and the mobile code can not proceed to execute its task.

This would threaten the autonomy of mobile code. The solution as discussed in section 3.5.1 is a proxy repository in the *neighborhood* of the mobile code. The general term *neighborhood* is introduced because the most favorable location of the repository which supports the autonomy of the mobile code at the same time keeping down the overhead, is application dependent. I.e. the nature of the route of the mobile code – assuming a multi hop agent – determines the optimal location of a repository.

## 5.2 Example of Adaptation Process

For illustrating the role of the adaptation elements described above this section presents an example for executing the method of an adaptable class. The adaptable class is described by a functionality interface `IMemory` which declares the method `getPhysicalMemory()` retrieving the size of the installed memory of a host. The functionality interface is supported by a set of implementation classes for different environments.

The adaptor class `IMemory_Adaptor` is generated out of the functionality interface and used in the core for accessing the implementation classes. It is assumed that the core is running on a *x86* host with *Windows NT* and no implementation class is yet loaded by the adaptor class.

Following steps are performed by the adaptation mechanism for executing the implementation method `getPhysicalMemory()` provided by a suitable implementation class (s. figure 5.1).

Step 1 The core invokes the method `getPhysicalMemory()` offered by the adaptor class `IMemory_Adaptor`.

Step 2 Since no implementation class is yet available, the adaptor class contacts the adaptation class to load the suitable implementation class supporting the functionality interface `IMemory`.

Step 3 The class loader of the adaptation class requests the implementation class name from the context awareness mechanism.

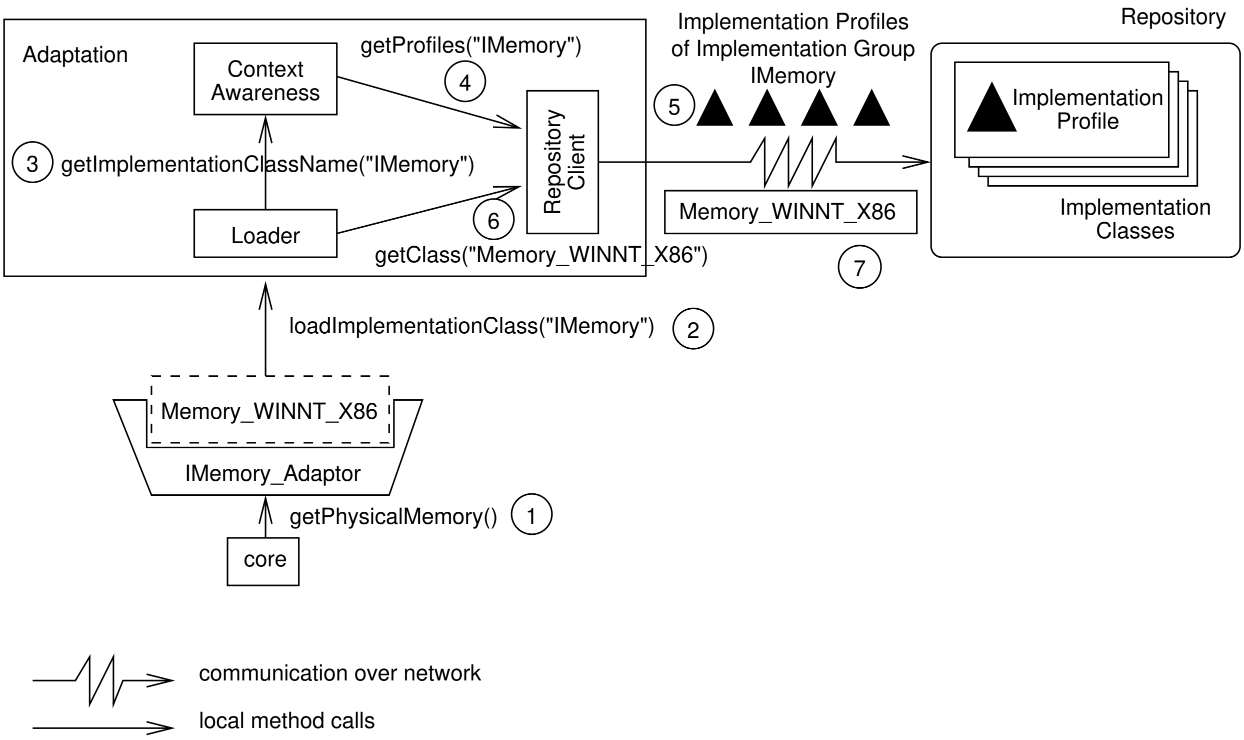


Figure 5.1: Steps of adaptation process

Step 4 For resolving the right implementation class name the context awareness needs the profiles of all implementation class belonging to the implementation group `IMemory`. Therefore the context awareness retrieves the profiles from the repository through the repository client.

Step 5 The repository collects all profiles from the implementation group `IMemory` out of the implementation classes and sends them back to the context awareness via the repository client. The context awareness obtains the profiles, executes them and returns the right implementation class name to the loader.

Step 6 The loader requests the suitable implementation class `Memory_WINNT_X86` from the repository via the repository client.

Step 7 The repository sends the implementation class to the loader through the repository client. The loader returns the implementation class to the adaptor class, which instantiates the class and executes the method `getPhysicalMemory()`.

If further methods of the class `IMemory_Adaptor` declared in the functionality interface `IMemory` are invoked by the core, the adaptor class can directly delegate the method calls to the implementation class `Memory_WINNT_X86`. A new adaptation process is necessary if the core moves to a new host.

### 5.2.1 Implementation of the Prototype

The proposed concept is realized in a prototype. The sample application is the configuration of web browsers by a mobile agent as described in section 3.1. The implementation uses the Voyager agent system platform [Obj00] as infrastructure for the mobile agent. The adaptation framework is written in pure Java. For the configuration of web browsers running on MS-Windows NT the access to the registry, the MS-Windows configuration data base the Win32 API is used, packaged in dynamic shared libraries for MS-Windows (DLLs).

## 5.3 Results

### 5.3.1 Transparency of Adaptation

As discussed in section 4.1.2 the proposed methodology does not offer total transparency to the application.

Adaptors hide adaptation up to certain limit, but there are still two aspects affecting the application:

- deployment of adaptors
- changed semantics of method invocations

Though the adaptor concept is a means to enhance transparency, the adaptors themselves lead to a different way of programming than needed for static customization.

Invocation of adaptable methods differs from ordinary local method calls, because an invocation of an adaptable method involves a selection of an implementation class requiring communication over the network with the repository. This implies two additional exceptions: The communication with the repository may fail due to a network error and the selection of an implementation class may fail because of a missing suitable implementation class for the current environment. These exceptional cases are mapped to one exception corresponding to the generic Java *class not found exception*.

### 5.3.2 Runtime Overhead

As presented in chapter 4.2 the implementation of the prototype using dynamic adaptation has a higher runtime than the equivalent version using static customization. This can be explained from a tradeoff between flexibility and performance as it occurs in many engineering disciplines. The runtime overhead caused by adaptation loses its effect on the overall runtime of the application, if it is applied to classes providing methods with a noticeable higher runtime than needed for the adaptation.

### 5.3.3 Breakeven Point of Adaptation

The essential criteria for the deployment of adaptation discussed in this work is the ratio of the size between the application-independent core and application-specific implementation classes. Adaptation is advantageous when the size of the environment dependent implementations is greater than the environment independent core.

## 5.4 Future Work

The current implementation of the repository has the drawback that instances of all implementation classes must be hold in the memory in order to get the according implementation profiles. This is sufficient if only a small number of implementation classes are needed as in the case of the sample application. Since a strength of dynamic adaptation is the gain of bandwidth in the case of a high number of implementation classes with a big size, the repository of the current implementation may become a bottleneck. The solution may be the loading and instantiating of each implementation class at start-up time of the repository. After the startup the separated profiles are saved only.

A further improvement concerning transparency would be the implementation of an adaptor generator which generates Java byte code during runtime and not Java source code as in the prototype implementation.

# Bibliography


[Ado85] Adobe System Incoporated. *PostScript Language Reference*, 1985.

[ADOB98] Gregory D. Abowd, Anind Dey, Robert Orr, and Jason Brotherton. Context-awareness in wearable and ubiquitous computing. *Virtual Reality*, 3:200–211, 1998.

[AW97] Noriki Amano and Takuo Watanabe. Lead: A language for dynamically adaptable applications. In *International Technical Conference on Circuits/Systems, Computers and Communications (ITS-CSCC'97)*, July 14-16 1997.

[BPW98] Andrzej Bieszczad, Bernard Pagurek, and Tony White. Mobile agents for network management. *IEEE Communications Surveys*, 1(1), 1998.

[Bri98] Bright Star Engineering. *ipEngine-1 Hardware Reference Manual*, 1998.

[DH88] Andrew Duncan and Urs Hölzle. Load-time adaptation: Efficient and non-intrusive language extension for virtual machines. Technical Report TRCS99-09, University of California, Santa Barbara, June 1988.

[Eng97] Robert Englander. *Developing JAVA Beans*. O'Reilly, 1 edition, 1997.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. RFC 2616, IETF, June 1999.

[FKK99] Metin Feridun, Wilco Kasteleijn, and Jens Krause. Distributed Management with Mobile Components. Technical report, IBM Zurich Research Laboratory, Rueschlikon, Switzerland, 1999.

[Fla97] David Flanagan. *Java in a Nutshell*. O'Reilly, second edition, May 1997.

[FPV98] Alfonso Fugetta, Gian Pietro Picco, and Giovanni Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):352–361, May 1998.

[GFP99] Thomas Gschwind, Metin Feridun, and Stefan Pleisch. ADK - Building Mobile Agents for Network and Systems Management from Reusable Components. Technical report, IBM Zurich Research Laboratory; Distributed Systems Group TU Wien, Rueschlikon, Switzerland, 1999.

[GK97] Michael Golm and Jürgen Kleinöder. Metajava – a platform for adaptable operating-system mechanisms. In *11th European Conference on Object-Oriented Programming (ECOOP '97) – Workshop on Object-Orientation and Operating Systems*, Jyväksylä, Finland, June 10 1997.

[Gra98] Mark Grand. *Patterns in Java, Volume 1, A Catalog of Reusable Design Patterns Illustrated with UML*. John Wiley, September 1998.

[Hei99] George T. Heineman. An evaluation of component adaptation techniques. In *International Workshop on Component-Based Software Engineering*, May 17–18 1999.

[KH98] Ralph Keller and Urs H"olzle. Binary code adaptation. In *12th European Conference on Object-Oriented Programming (ECOOP '98)*, Brussels, Belgium, July 20–24 1998.

[LB98] Sheng Liang and Gilad Bracha. Dynamic class loading in the java virtual machine. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, 1998.

[Lia99] Sheng Liang. *The Java Native Interface : Programmer's Guide and Specification.* Addison-Wesley, June 1999.

[LO98] David Lange and M. Oshima. *Programing and Deploying Mobile Agents with Java.* Addison-Wesley, 1998.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 2 edition, 1999.

[MKKM98] Nelson Minar, Kwindla, Kramer, and Pattie Maes. Cooperating mobile agents for mapping networks. In *First Hungarian National Conference on Agent Based Computing*, Cambridge, MA, USA, May 1998.

[Nob00] Brian Noble. System support for mobile, adaptive applications. *IEEE Personal Communications*, pages 44–49, February 2000.

[Obj00] Objectspace. *Voyager ORB 3.3 Developer Guide*, 2000.

[OH98] Robert Orfali and Dan Harkey. *Client/Server Programming with JAVA and CORBA.* John Wiley, 2 edition, 1998.

[Ple99] Stefan Pleisch. State of the art of mobile agent computing - security, fault tolerance, and transaction support. Technical report, IBM Zurich Research Laboratory, June 1999.

[SAW94] Bill N. Schilit, Norman Adams, and Roy Want. Context-aware computing applications. In *Proceedings of the Workshop on Mobile Computing Systems and Applications*, December 1994.

[ST93] Mike Spreitzer and Marvin Theimer. Scalable, secure, mobile computing with location information. *Communications of the ACM*, 36(7):27, July 1993.

[STW92] Bill N. Schilit, Marvin Theimer, and Brent B. Welch. Customizing mobile applications. In *Proceedings of the USENIX Symposium on Mobile and Location-independent Computing*, pages 129–138, August 1992.

[Sun99] Sun Microsystem, Inc. *Java Remote Method Invocation Specification*, December 1999.

[TSS+97] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A survey of active network research. *IEEE Communications*, 35(1), January 1997.

[WHFG92] R. Want, A. Hopper, V. Falcao, and J. Gibbons. The active badge location system. *ACM Transactions on Information and System Security*, 10(1), January 1992.