University of Groningen **Faculty of Mathematics and Natural Sciences**

Department of Computing Science

A Coordination Component Framework for Open Distributed Systems

Sander Tichelaar

Supervisors:
Prof.drs. C. Bron
Department of Computing Science
University of Groningen

Prof.dr. O. Nierstrasz
J.C. Cruz
Institute of Computer Science
and Applied Mathematics
University of Berne

May 1997

RuG

Abstract

We have investigated software development for open distributed systems in order to make this development easier. Easier in the sense that software parts will be better reusable, more flexible and better maintainable. The hardest part is to address evolution of these systems because not all application requirements can be known in advance. In particular we have investigated the coordination aspects of open distributed systems. Coordination technology addresses the management of interaction of software agents in a distributed or parallel environment and, therefore, typically describes architectural aspects of a system. To reach the goal of easier software development we have applied a component oriented approach: generic coordination solutions are provided as generic architectures with black-box components. Applications are constructed using these architectures and composing and parameterizing these generic components. In this way we make the interaction part of a system reusable and flexible. The architecture of the system is also made clearer and therefore easier understandable.

A prototype coordination framework and a set of sample applications that are representative for open distributed systems and that use this framework, have been developed in the concurrent object-oriented programming language Java. We show that, using our component-oriented approach, we gain reusability, flexibility and provide clear architectures of applications. A major problem, however, concerning the genericity of components, is the application dependent information that may be needed by a coordination solution: the genericity of the solution is strongly dependent on the possibility to separate this information from the generic solution.

Samenvatting

In dit afstudeerproject hebben wij software ontwikkeling voor open gedistribueerde systemen bestudeerd. Het doel is deze ontwikkeling te vergemakkelijken door programmadelen beter herbruikbaar, flexibeler en beter onderhoudbaar te maken. Het moeilijkst is het om evolutie van dit soort systemen voor elkaar te krijgen, want toekomstige systeemeisen zijn niet allemaal te voorspellen. In het bijzonder hebben wij gekeken naar de coördinatieaspecten van open gedistribueerde systemen. Coördinatietechnologie houdt zich bezig met het organiseren van de interactie van software agents in een gedistribueerde of parallelle omgeving en beschrijft, daarom, de typische aspecten van de architectuur van een systeem. Om tot een verbeterde software ontwikkeling te komen hebben we een componentgeoriënteerde aanpak gekozen: generieke coördinatieoplossingen worden aangeboden als generieke structuren met black-box componenten. Applicaties kunnen geconstrueerd worden door gebruik te maken van deze structuren en door het samenstellen en parametriseren van de generieke componenten. Op deze manier maken we het coördinatiegedeelte van een applicatie herbruikbaar en flexibel. De architectuur van een systeem wordt op deze manier ook duidelijker gemaakt en daardoor makkelijker te begrijpen.

Wij hebben een prototype van een coördinatieframework gemaakt en enkele voorbeeldapplicaties die representatief zijn voor open gedistribueerde systemen en die gemaakt zijn met behulp van dit framework. De gebruikte programmeertaal is de concurrent objectgeoriënteerde taal Java. Wij laten zien dat we met onze componentgeoriënteerde aanpak aan herbruikbaarheid en flexibiliteit winnen, en dat we duidelijk de structuur van een applicatie kunnen tonen. Het grootste probleem dat we tegenkwamen bij het ontwikkelen van generieke componenten is de applicatieafhankelijke informatie die een coördinatieoplossing nodig kan hebben: de genericiteit van een oplossing is sterk afhankelijk van de mogelijkheid om deze informatie los te koppelen van de generieke oplossing.

Preface

This report is a result of the project I did to get my Master's Degree in Computing Science at the University of Groningen. The project was a perfect match between my interests in software engineering, object-orientation and my wanting to do a project "somewhere abroad": it was a project in the area of component-oriented software development, carried out in the Software Composition Group of the University of Berne.

First of all, I want to thank Professor Bron for supervising my work and for giving me the opportunity to do my project somewhere far away. And even there he managed to drop by to see how I was doing.

Secondly I want to thank the people at the SCG. Of course, Professor Oscar Nierstrasz, head of the group, for giving me the opportunity to come to Berne and supervising me during the project. Juan Carlos Cruz, also supervising me, for the work we did together, and the careful reading of initial drafts of this report. Theo Dirk Meijler and Serge Demeyer for their valuable remarks (in Dutch, of course!) about (the organization of) my work. Furthermore I want to thank all the other group members for their help and for the good time I had during my stay in Berne.

Finally I want to thank some of my fellow students during both my physics and computer science studies. And, of course, my friends and family. Without them I wouldn't be where I am now.

Sander Tichelaar
May 1997

Contents

Chapter 1

Introduction

Nowadays we cannot view software systems as being closed and proprietary anymore. Most modern software applications are open in terms of topology, platform and evolution: they are built on ever expanding networks, on heterogeneous platforms and they are subject to constantly evolving requirements[39]. So modern applications are just parts of distributed, inter-operable and flexible software systems. Distribution and interoperability are relatively easy to obtain, because these requirements are known at design time. Flexibility is the most difficult to meet, because not all application requirements can be known in advance[26].

The problem we address in this thesis are the difficulties that exist in the development and evolution of open systems as described above. We particularly address the coordination aspects of these systems.

In general systems can be described as computational parts that interact with each other. These computational parts have to be coordinated to enable them to work together:

Systems = Computation + Coordination

Coordination, therefore, can be viewed as the management of this interaction[17], as the "glue" between the computational parts[11]. Coordination describes typically parts of the application architecture, because it describes which activities work together and how they work together. Most of the work done on coordination so far has focused on the development of particular languages that realize a particular paradigm for realizing coordination. Examples of these languages are Linda[5] and Gamma[4]. Coordination problems, however, are not always well-suited to a particular paradigm[6]. What coordination languages don't address are reusable abstractions at a higher level than the basic mechanisms and paradigm supported directly by the language.

The approach we take to tackle the problem of development and evolution of open distributed systems, particularly their coordination aspects, is a component-oriented approach. Component-oriented approaches have become increasingly popular in the last couple of years: software should be developed using flexible and reusable software abstractions that can be used to compose applications. Although it is an old idea to use "pre-fabricated" and reusable "software components"[25], it has become an issue again with renewed interest in object-orientation and the introduction of component-based software development tools like Visual Basic[27] and Delphi[31] (*[Components:] they're baaack!*[28]).

Our approach is focused on building generic coordination components. These components realize generic solutions to standard coordination problems. They can be specialized and parameterized to solve specific coordination problems. With these components we provide explicit separation of coordination and computation, so facilitating reuse and evolution of coordination aspects. And as coordination aspects describe typically parts of the application architecture, we make the architecture more explicit and manipulable. A description of which components are used and how they are put together and parameterized, provides a high-level description of what happens where in a system[1]. This makes a system easier to understand and easier adaptable to new requirements[26].

Object-oriented programming languages(OOPLs) go a long way towards supporting components. Objects hide their implementation and there are numerous object oriented design patterns that exploit the possibilities of run-time object composition[9]. There is, however, still to do a lot in this area to come to a more rigorous and complete approach to component-oriented software development[26]. Typical problems with the OO paradigm with respect to building software components are:

- existing OOPLs emphasize reuse by programming new object classes that extend existing ones and not by composing different objects or object-parts together[26].
- OO properties can be hard to use in parallel without violating the advantages they offer[30]. A typical example is inheritance and encapsulation: when using inheritance, a programmer may need to know implementation details of a superclass, so violating encapsulation.
- OO leaves the architecture of a system mostly implicit. The composition of objects is typically hidden in the implementation of the objects themselves[26]. A structural description, however, is of use during design, documentation and subsequent maintenance of a system[18].

In this project we tried to overcome these problems by imposing extra requirements on the way a system is designed. The project is a pilot project in applying the component idea to coordination using an existing concurrent object-oriented programming language.

To validate our ideas we developed a prototype coordination framework in Java[12], an object-oriented programming language particularly well-suited to modeling software entities in a distributed environment. This language provides low-level network communication abstractions and some basic synchronization primitives, that are useful to build coordination abstractions. We tested the usability of the framework by applying it to a set of sample applications that are characteristic of open systems.

In part I of this thesis we discuss open distributed systems and coordination (our problem domain). We also present our component-oriented approach. In part II we present an analysis of our sample applications with respect to coordination problems in open distributed systems and we describe our experiences while developing the sample applications and the framework. We show that we indeed gain reusability, flexibility and that we provide clear architectures of the developed applications. But, as this project was a first try to develop a coordination framework, we only have some preliminary results. We end with a summary and discussion of these results in part III.

[1]As also promoted by the configuration language Darwin[22]

Part I

Background

Chapter 2

Problem domain

In this chapter we describe the problem domain of our project. This problem domain is open distributed systems and coordination. We will first introduce these two notions and after that we will discuss coordination problems in open distributed systems.

2.1 Open distributed systems

In this section we first start with a definition of distributed systems and next, we introduce the additional requirements for open systems.

2.1.1 Distributed systems

Distributed systems are systems where different parts of such a system are geographically separated. We call these parts *entities* (other common names are "active entities", "active objects", "agents", "actors", etc.). These entities are physically distributed, but interconnected. They run, for instance, on different computers in a network, but they have to exchange information to be able to work together.

Reasons for having distributed systems as stated above are[38]:

- Information exchange: Different entities may need information from each other.
- Resource sharing: Clients make use of common resources, for instance a central database.
- Increased reliability through replication: If some nodes of a system may fail, other nodes that still operate correctly, can take over the tasks of the failed ones.
- Increased performance through parallelization: If a number of tasks are executed in parallel, the overall performance of a system can be better than if these tasks were executed sequentially.
- Simplification of design through specialization (expressiveness): Different parts of a system can do a specialized task, maybe even on specialized computers in a network.

We see that the entities are separated but interdependent: they are designed to achieve a common goal. They shouldn't be too interdependent, because this would violate the advantages of distribution that we have stated above. So the different entities form a coherent, but loosely coupled system which provides an integrated computer facility: their common goal.

This leads to the following definition of Distributed Systems:

> *A Distributed System is a collection of loosely coupled entities in a distributed environment, working together to achieve a common goal.*

Distributed systems have a set of characteristics that distinct them from non-distributed systems. In the ISO draft for a reference model of distributed processing[14] the following characteristics of distributed systems are mentioned:

- remoteness: follows clearly from the distributed nature.
- concurrency: any activity in a distributed systems can be executed in parallel with other activities.
- lack of global state: it is impossible to determine the state of a distributed system precisely, because a node in a system only knows its own state.
- partial failures: parts of a system can fail independently from other parts of the system.
- asynchrony: due to possible differences in execution speed of different activities the system is non-deterministic.

2.1.2 Open distributed systems

Most modern applications must satisfy some additional requirements over the ones we mentioned in the previous section. They have to act on ever expanding networks, on heterogeneous platforms and they are subject to constantly evolving requirements. Applications that can deal with the above requirements are called *open systems*[39].

In open systems we, therefore need apart from distribution and interoperability, a great deal of flexibility. Flexibility in the topology of a network: network architectures and, for instance, the number of clients can change. Flexibility in platform: applications have to run on and communicate with different platforms. The most difficult kind of flexibility is flexibility needed to cope with evolution, because not all application requirements can be known in advance[26].

In the same ISO draft[14] the following additional requirements are mentioned for open distributed systems:

- heterogeneity: systems have to cope with different and changing hardware, operating systems, communication networks and protocols, etc.
- autonomy: the various management or control authorities an organizational entities are autonomous.
- evolution: systems have to cope with changing application requirements.
- mobility: activities and data may be moved over a network.

2.2 Coordination

Coordination has to do with interaction. Whenever active entities interact they have to act in a *coordinated way* to get to a result. When people want to meet, they have to be at the same time at the same place, otherwise the meeting will fail. Or, when multiple users want to read and write in a database, the access to the database has to be coordinated in order to keep the database consistent.

So whenever multiple entities are involved we need some kind of coordination to enable them to work together and to resolve conflicts between them. One definition of coordination is given by Malone and Crowston[21] in an interdisciplinary study of coordination. This study covers fields from economics and organizational theory to computer science. They say that

> *Coordination is the act of managing dependencies between activities.*

This is consistent with the intuition that, if there is no interdependence, there is nothing to coordinate. Kielmann[17] says the same with other words, namely that coordination is the managing of inter-agent activities of agents collected in a configuration. He doesn't say, however, that these inter-agent activities have to do with dependencies. A broader definition is given in [7]:

> *Coordination is the organization of a group of entities in order to improve their collective results.*

This definition takes all organization of entities into account, even when there are no dependencies between them. An example of organization without dependencies is the organization of entities, when global constraints exist, like imposed conditions on the way in which solutions must be implemented by entities[1]. In this thesis, however, we focus on the first definition.

The first definition implies that coordination is needed whenever there are some interdependencies between activities. Malone and Crowston[21] present the following list of interdependencies:

- Shared resource: a resource is used by multiple activities.
- Prerequisite: an activity must be completed before another can begin.
- Transfer: an activity produces something that is needed by another activity, and this "something" should be transferred from one activity to another.
- Usability: whatever is produced by an activity should be usable by the activity that needs it.
- Simultaneity: some activities need to occur (or cannot occur) at the same time.
- Task/Subtask: a task is divided into a set of subtasks that can be executed by different activities.
- Group decisions: decisions are taken collectively by a group of entities.

Several generalizations and specializations are possible, for instance concerning aspects like number of activities involved in a dependency (e.g. we can define a Multiple-Prerequisite dependency, where some activities need to be completed before others can begin) and time (e.g. we can define a Delta_Time-Prerequisite: an activity must begin a certain time interval after another activity has ended)[6].

There are several ways of dealing with the coordination problems that arise in case of the interdependencies stated above. Mintzberg[29], for instance, considers three fundamental coordination styles:

- Mutual adjustment: This occurs whenever two or more entities agree to share resources to achieve some goal. Entities must exchange information and make adjustments in their behaviour, depending on the behaviour of other entities. In this form of coordination no entity has prior control over the others.

[1]Even in this case we can view these global constraints as interdependencies. These interdependencies, however, are not related to the task of the entity, but to the environment.

- Direct supervision: This occurs when two or more entities have already established a relationship in which one entity has some control over the others. The prior relationship is commonly established by mutual adjustment. In this form of coordination the supervisor controls the use of common resources and prescribes certain aspects of the behaviour of its subordinates
- Standardization: This occurs when entities have to follow pre-established standard procedures in a number of situations. In this form little coordination is needed, until the procedure itself needs to change[2].

In the book “How to write parallel programs” by Carriero and Gelernter[2] it is shown that different forms of parallelism, and therefore different kinds of interaction, may favor different paradigms for interaction. The authors discuss three different forms of parallelism in parallel programs (and in distributed systems): result parallelism, specialist parallelism and agenda parallelism. With *result parallelism* a problem is divided into parts and there are many workers that produce a piece of the result. This kind of parallelism naturally maps to live data structures: processes are represented by their results. Each data element is implicitly a process which will turn into a (sub)result data object when the process terminates. With *specialist parallelism* every parallel activity has its own competence. This kind of parallelism is a good match to message-passing: each activity can be on a network node and messages implement communication over edges. *Agenda parallelism* is a kind of parallelism where the work is organized as an agenda of activities. Workers are generalists that grab a task that is needed to be done at that moment. This maps naturally onto a (distributed) shared data structure: data elements are accessible through the whole (distributed) system, so every activity can access and process these elements whenever needed.

We see, different problems need different solutions. And there are different ways of solving problems. In our work we keep all possibilities open. Depending on the needs of our system we can have centralized coordination, non-centralized coordination, we can use message-passing or generative communication, or whatever is needed to solve a specific problem in a specific system.

2.3 Coordination problems in open distributed systems

For the identification of coordination problems in open distributed systems we use the definition of Malone and Crowston in section 2.2. This definition states that coordination is the management of dependencies between activities. We, therefore, look at dependencies in open distributed systems and it appears that there are many dependencies (and thus coordination problems) that appear over and over again in these systems. A first list of these problems is presented in [6]:

- simultaneity constraints between activities: activities are dependent, because they need, or cannot, occur at the same time. A well-known example of this kind of constraint is a shared resource (e.g. only one activity can write in a database in order to keep it consistent).
- execution ordering between activities: activities are dependent, because they need to appear in a certain order (e.g. a file must be opened before write operations can be done).
- transfer of information between activities: activities are dependent, because they need information from each other and this information has to be transferred between them (e.g. when computing the topology of a network).

[2]This is, however, viewed from a human management point of view: a manager doesn’t have to do anything unless a standard procedure changes. In computer systems, the implementation of the standard procedure will be viewed as the coordinating entity.

- simultaneity constraints between activities: activities are dependent, because they need, or cannot, occur at the same time (e.g. only one activity can write in a database in order to keep it consistent).
- task/subtask dependencies: in computer systems these kind of dependencies are usually determined at design-time, when the programmer decomposes a goal into subgoals. Dynamic goal decomposition can be found in multi-agent systems, and we can also think of dynamic decomposition for reasons of load balancing.
- group decisions: activities are dependent, because they need each other to take some decision (e.g. a new main server has to be chosen in a group of servers, when the former main server is down).

This list is not intended to be exhaustive. It is a first set that we have identified. We can use it to analyze particular open systems and we can always extend it when we find other dependencies in open distributed systems.

As an example we go into more detail for transfer of information and the access to shared resources.

Transfer of information

We can view information transfer dependencies as producer/consumer relationships: one activity produces some information that is used by another activity. We need a coordination solution to control this transfer of information[3]. This coordination solution must: take care of the physical transfer of the information from one activity to another, control their synchronization, and, in case of replicated transfer (multi-cast, broadcast, etc.), control the replication and transfer of information and, if needed, guarantee the atomicity (all or none of the activities will receive the information) and the order of arrival of the information[6].

In figure 2.1 a simple one-to-one transfer problem is shown. The solution should take care of the requirements stated above. As an extra feature it could also provide buffering of information.

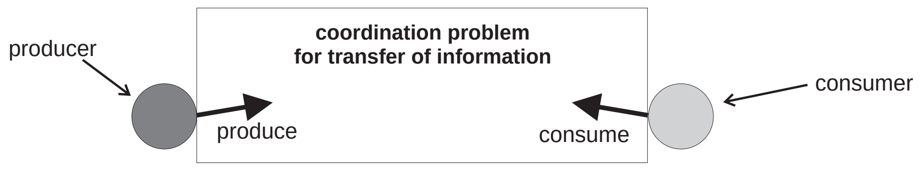


Figure 2.1: One-to-one transfer dependency problem

Access to shared resources

Multiple entities may need access to the same resource. A solution to this coordination problem should take care of serializing concurrent requests, or, at least, take care that no requests that are harmful together, will enter the resource at the same time (e.g. this is the case when there is a readers/writer access to shared resource. Writers have to enter the resource alone, readers can access it with more

[3]At this point it is open if this solution is implicit in the code or taken care of by an operating system, or that we use a special coordination component to solve our problem.

than one at a time). The solution also should control fairness and access rights and take care of possible hardware and software failures[6].

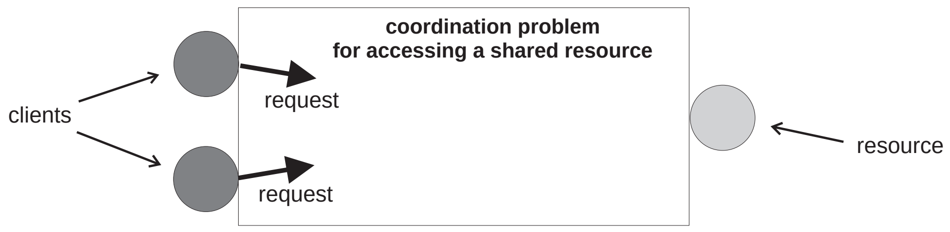


Figure 2.2: Shared resource access problem

Chapter 3

Approach

3.1 A component-oriented approach to coordination

In this project we have explored a component-oriented approach: component software should be written as black-box abstractions that can be composed and parameterized to construct an application. Particularly we have had a look at coordination aspects in open distributed systems. These (non-functional) aspects are typically programmed for specific classes[26] and, therefore, difficult to reuse and not very flexible. We have tried to capture solutions to coordination problems in generic software abstractions to improve reusability and flexibility of these aspects and to support an explicit representation of a system's architecture.

The need for reasoning about the architecture of software systems, especially when these systems are large and complex, is, among others, stressed by Garlan and Shaw[10] and Perry and Wolf[32]. Explicitness of structure can be helpful for design, documentation and maintenance[18]. A clear architecture makes a system easily understandable: it is clear what happens where in the design, and thus, in case of maintenance, where the system should be adapted to new requirements. One way of dealing with the problem of interacting components is the use of a configuration language like Darwin[22]. This language describes programs as a set of component instances and their interconnections. It allows the specification of both static structures fixed during system initialization and dynamic structures which evolve as execution progresses[23]. Explicitness of architecture may seem to violate transparency principles. But on every level of an application a clear picture of the underlying component structure helps in understanding that part of an application. This, of course, without violating the transparency of the underlying black-box components. We can make a comparison with imperative languages: a good choice of modules and procedures provides on every level of the application a clear description of what happens on that level, but keeps the actual implementation transparent.

Our approach is aimed at developing components to provide coordination solutions. These solutions typically define (parts of) architectures of systems. In this way we do not only make the structure of an application clearer (as Darwin does), but also make these non-functional aspects of an application reusable and flexible (as components in general should do). We introduce the concept of components and component framework in section 3.2.

We have chosen an object-oriented programming language (OOPL) as the implementation language for our components, because OO languages go a long way towards supporting components: objects provide encapsulation of data and services by hiding their implementation details. And several design patterns exploit the possibilities of object composition to gain reusability and flexibility[9].

Composition is also provided by class composition mechanisms like inheritance, templates and mixins[1]. There are, however, some problems using OO for building components. We discuss these problems in section 3.3.

3.2 Components and Frameworks

For a programming language to support component-oriented software development, it must cleanly integrate both the *computational* and the *compositional* aspects of software[30]. These aspects, however, are not always integrated in a straightforward way due to interference of different object-oriented features (see section 3.3 for a discussion). Computational requirements will be fulfilled anyway (otherwise a system makes no sense). The point is how these computational requirements are organized: how are the computational parts put together.

Although the OO paradigm promised to provide reuse, it turned out that OO doesn't do this in itself. Reusability of (parts of) a system is only reached, if this aspect is taken into account during the whole software development process[16]. The same applies for composability. Unlike, for instance, in functional programming where it is easy to construct a new function by combining two or more others, composability is not well supported in object-oriented technology. Therefore a similar idea as for reusability is proposed for developing composable software: Software development should be (component) framework-driven. All phases of the software life-cycle including requirements collection and specification should be aimed at developing patterns and components formalized within a framework[26].

3.2.1 Components

In [26] a component is defined as

a component is a static abstraction with plugs

An *abstraction* can more or less be any useful abstraction you can think of: an interface, an object, a class, a template, a type, a function. The implementation details of the encapsulated structure are hidden. "*Static*" means that a component is a long-lived entity that can be stored independently of the applications it is a part of. "*With plugs*" means that the interaction of the component with other components is well-defined. Plugs can for instance be parameters, ports, references to objects, etc.

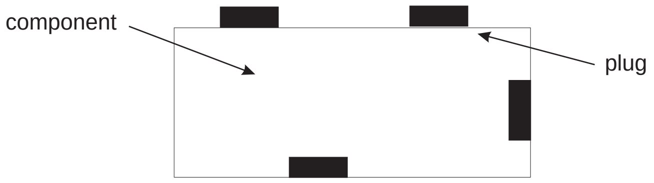


Figure 3.1: A software component

This definition doesn't say anything about reusability and flexibility. Developing components, however, has everything to do with these two aspects. We are developing components in the first place to be reusable and flexible to ease software development. Reusability and flexibility lead to some extra demands on components: they have to be generic and (re)configurable. "*Generic*" means that the component will be applicable in a range of common problems. So components must provide a common solution, which can be configured for specific use. Flexibility demands that components be easily adaptable to new requirements, for instance for maintaining and adapting evolving software systems in a simple way.

A last point about components is that encapsulation of abstractions also supports explicitness of architecture.

This leads to the following description of components:

> *A component is a generic black-box abstraction which is (re)configurable and composable by plugs*

Applications can then be viewed as compositions of parameterized components (see figure 3.2).

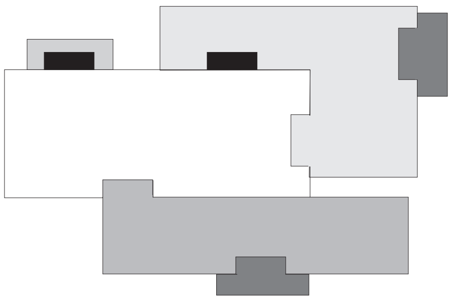


Figure 3.2: An application composed with components

This is, however, a static view of applications. Applications can also be viewed as a dynamic assembly of cooperating and communicating ”entities” (like in the configuration language Darwin[22]). This is typically the view of an application at run-time. The boundary between these two views is not always very clear. Nowadays, applications can be (re)composed using means as dynamic loading or dynamic method lookup. Therefore we have to distinguish between components at design time, components at run-time and maybe components that exist in both these views. Another consequence is that a composition can change in (run-)time.

3.2.2 (Component) Frameworks

We saw that object-oriented technology doesn’t provide reuse by itself. It depends on the way one uses the available technology. In [26] is shown that all approaches to develop open, adaptable systems, in some way, are based on component frameworks. In [16] frameworks are defined as

> *A framework is an abstract object-oriented design, where every major part of the design is represented by an abstract class. Usually there is a library of subclasses that can be used as components in the design.*

OO frameworks are more than just class libraries. They provide a generic architecture (the abstract design) and mostly act as the coordinating and sequencing application activity.

Commonly specific behaviour is induced by adding or adapting methods in subclasses of the classes provided by the framework. We call frameworks that use this method *white-box frameworks*, because typically the implementation of the superclasses must be understood to use them. A consequence of this way of specialization is, that many new subclasses have to be written and that these subclasses will always be dependent on their superclass(es). This makes it difficult for a new programmer to understand an application and thus can make an application hard to adapt.

Black-box frameworks, on the other hand, make use of components with a particular interface (as described in the previous section). The user of the framework only has to understand these interfaces to be able to use the components. An application is constructed by plugging the black-box components into the generic architecture. Where the former way is more flexible (functionality is quickly adapted by programming a new subclass), the latter is easier to use (components are known only by their interfaces, and easily interchanged if their interface is the same). And with black-box frameworks it is clearer what happens where: the structure of the solution is explicit.

3.3 Components and OO technology

Object-oriented technology supports components to a certain degree. Objects provide encapsulation of data and services by hiding their implementation details. And several design patterns exploit the possibilities of object composition to gain reusability and flexibility[9]. But there are some problems concerning component development in standard object-oriented languages. These problems have to do with the focus on white-box reuse[26] and the fact that several OO features appear hard to integrate[30]. We will go into more detail on these problems below.

A widely used method to gain reusability and flexibility is the use of design patterns. Design patterns provide general solutions to common software design problems. A good pattern can be used over and over again to solve the same kind of problem in different settings. Because patterns capture mostly solutions that have been developed and have evolved over time, the solutions tend to be more flexible, modular, reusable and understandable than ad hoc solutions (example: Strategy pattern)[9]. Patterns aren't, however, strongly supported by object-oriented systems. It is not possible to make patterns explicit in a design and there aren't built-in mechanisms to enforce correct use of patterns.

OO focuses on white-box reuse

Reuse in object-oriented systems is mostly obtained by inheritance: new classes are programmed by specializing and extending superclasses[26]. In this way reuse is realized by programming new classes that extend other ones. This approach is inherently limited, because encapsulation of the superclass is violated and subclasses can be strongly dependent of superclasses. This kind of reuse is called "white-box" reuse: a programmer typically has to know implementation details of the superclass. The advantage of this approach is that it is flexible: all kinds of subtle behaviour differences can be obtained by slightly adapting classes.

"Black-box" reuse is preferable because it is easier to use and more robust: a programmer only has to understand the interface of the abstraction and is not able to write code that is dependent on implementation details of the abstraction. It is, therefore, somewhat less flexible than "white-box" reuse. With this kind of reuse whole abstractions are reused without any additional classes to program. Behaviour is adapted by parameterizing the abstraction with parameters, like just values, objects or even types (e.g. in C++-templates).

Another consequence of the fact that object-oriented approaches do not focus on designing composable abstractions is that they don't provide support for explicitly representing the architecture of an application. Links are often hidden in the extension code. This makes a system harder to understand and more difficult to adapt. Design patterns often try to make class structures as clear as possible (example: Strategy pattern[9]). But, as we already mentioned, patterns are not well supported by current OO systems. As patterns can make a structure more explicit, it is not possible to make them explicit in a design. When using components links and dependencies between components must be explicitly specified, thus making the architecture explicit[26].

Interference of OO properties

Wegner has made a classification of object-based programming languages [40]. He proposes the following categorization of languages along with their "dimensions":

Object-based:	encapsulation and identification
Object-oriented:	+ classes + inheritance
Strongly-typed:	+ data abstraction and types
Concurrent:	+ concurrency and distribution
Persistent:	+ persistence + sets

Additionally another dimension is mentioned in [26], namely *homogeneity*: in a homogeneous object-oriented language, everything (within reason) is an object. So Smalltalk is a homogeneous object-oriented language whereas C++ is not.

These dimensions are said to be orthogonal, which in this case means that the different elements can be found independently in different programming languages. It appears that integrating the different dimensions is not trivial: they interfere in unexpected ways[1][30]. We saw in the previous section already that the use of inheritance violates object encapsulation because subclasses must typically be aware of implementation details of the superclass.

Similar problems exist with concurrency. Classes that use a concurrency mechanism are difficult to inherit from without knowing the concurrency details of the superclass. A subclass, for instance, needs access to a mutex in the superclass in order to synchronize its own methods with methods from the superclass. McHale[24] has shown that the cause of this problem lies in conflicts between inheriting sequential code and inheriting synchronization code. He proposes generic synchronization solutions to separate synchronization code from other code. In his thesis many examples can be found of generic synchronization policies that are independent of, but can be bound to, classes that need this synchronization. Another example of interfering features is concurrency and objects: they interfere, because objects that function correctly in a sequential environment, may not function in a concurrent setting.

More examples of interference can be given and they all come down to the same problem: the interference shown above, is a consequence of an inadequate client/supplier contract[30]. Classes, for instance, provide an interface to instances and one to subclasses and these two are not clearly separated.

[1]Orthogonality from the mathematical point of view means that there is *no* interference between dimensions! So, from that point of view, interfering orthogonal dimensions are a *contradictio in terminis*......

Part II

Experiments

Chapter 4

Sample applications

Our prototype coordination component framework should be useful for the development of coordination parts of open distributed systems. We need, therefore, a set of applications to develop, test and evaluate our framework. These applications should be representative for the problem domain. In this way we ensure that the solutions are solutions to actual problems and thus that our approach has the required capabilities.

In section 4.1 we discuss the criteria for our set of sample applications. In section 4.2 we discuss the sample applications we chose: we give a short description of the application and a list of coordination problems in these applications. At the end we categorize the solutions in order to do a first step towards generic solutions.

4.1 Criteria

The criteria for our sample applications are:

- They should be representative for open distributed systems: they should cover the main properties of open distributed systems.
- They should cover a range of coordination solutions: the framework is supposed to provide a set of coordination solutions. To develop and test a first set, the sample applications should cover the set of problems that require these solutions.
- There should be some overlap between the sample applications: equal or comparable coordination problems should appear in different applications in order to ensure that the given (implementations of) solutions be general and reusable.

4.2 Chosen sample applications

Out of a longer list of possible applications (based on, among others, [37]) we have chosen five sample applications that are representative of open distributed systems. We describe them shortly in section 4.2.1, we discuss the coordination problems that appear in these applications in section 4.2.2 and end with a categorization of these problems in section 4.2.3. We do this in the form of a matrix so we see not only which solution appears where (the second criterion), but also where the overlap of solutions in different applications (the third criterion) can be found.

4.2.1 Description of the sample applications

In this section we list the sample applications we chose for the development of our framework. Distribution in these applications is mostly obvious due to the distributed nature of the applications: parts of the system are geographically separated and/or clients need remote access to a service. Openness isn't inherent to the applications themselves. It is encountered in additional requirements we impose: they have to be able to cope with evolving network topologies, heterogeneity and other evolving requirements. Basically this comes down to extra flexibility requirements to a system.

Automated Teller Machine System: A system to support a network of ATMs shared by a consortium of banks. Every bank has its own account database. Teller machines are connected to a central server (one per bank). The teller machines, although owned by a specific bank, can serve clients from other banks as well. The application should be open in the sense that when a new teller machine is added, the system shouldn't be reconfigured or restarted. Different banks can also work with different platforms.

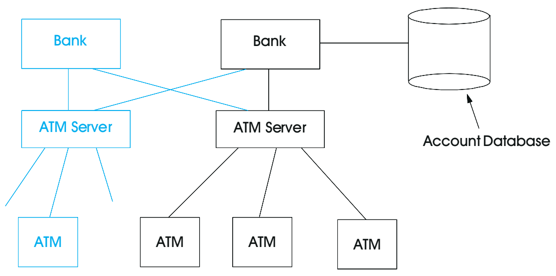


Figure 4.1: Automated Teller Machine System

Library Application: A system that keeps a database of all books in a library. The database should be accessible through a network for the following functions: searching for books and retrieving information about them (are they in the library, have they been lent, are they reserved). Also actions like reserving a book, registering a book as lent, updating the library (new books in, old books out, overruling reservations) are possible.

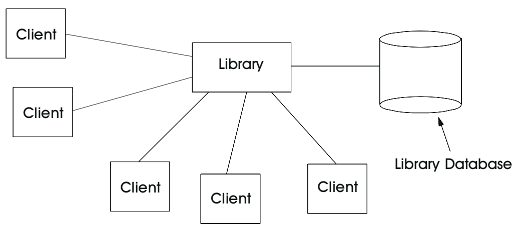


Figure 4.2: Library Application

Game Server: There is a central game server which provides means for communication between different players. If necessary it holds a central play-field or multiple play-fields for, for instance, one-to-one games. These play-fields can also be at the client side, in which case the server only provides communication.

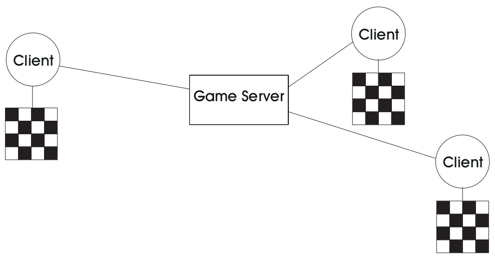


Figure 4.3: Game Server

Chat system: a couple of users at different sites in a network can send messages to each other that all users can see in a window.

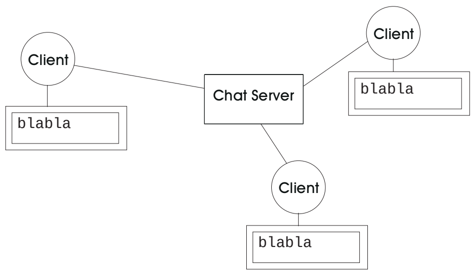


Figure 4.4: Chat System

Multi-user drawing program: We have chosen one program in the area of computer supported cooperative work(CSCW): a drawing program that can be used by multiple users at the same time. (Parts of) the drawing area are lockable so that no more than one user is drawing in an area.

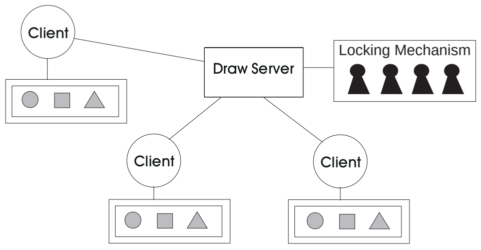


Figure 4.5: Cooperative drawing system

4.2.2 Coordination problems in the sample applications

In this section we list the coordination problems per sample application[1]. Some coordination problems will not appear in this list: internal communication (e.g. message passing within a client) and type-checking as usability constraint (this is also taken care of by the program environment itself). Also

[1]see also section 2.2 for a theoretical background of the categories.

time-out facilities for fault tolerance reasons are not mentioned, because they are inherent in every open distributed application.

Another thing not mentioned in the following list is: task/subtask dependencies (because there are none of them in these applications. See also the description of this dependency in section 2.3). Nor do we mention, that there can be centralized and non-centralized solutions.

Automated Teller Machine System:

transfer:	remote communication
	redistribution of ATM requests to different banks
shared resource:	account database access
	service access to bank and request redistributor
prerequisite constraints:	account, PIN, balance checks before deducting from an account
simultaneity constraints:	(readers/writer) access policy to account database
usability:	transformation of currency (according to some exchange rate)
replication:	replicated account databases
group decision:	choice of new replica coordinator between replica managers if coordinator is down

Library Application:

transfer:	remote communication
shared resource:	book database access
	service access to request manager
prerequisite constraints:	allowed actions on book depend on status of book
	allowed actions on books depend on status of user
	updating the library only if user is library master
simultaneity constraints:	(readers/writer) access policy to database

Game Server:

transfer:	remote communication
	two-way connection (between two players)
	multi-cast connection (1-to-N connection between player and other players)
	redirection/regrouping of communication requests (e.g. group two incoming requests from people who want to play, so that they can play a one-to-one game)
shared resource:	central play-field (depends on game)
	service access to server
prerequisite constraints:	conditions to start a game (e.g. at least two players)
simultaneity constraints:	access policy to central play-field (if there is one)
time constraints:	response time (especially if the game is real-time)
group decision:	to decide who starts the game

Chat System:

transfer:	remote communication
	multi-cast to all participants
shared resource:	server
	windows that display the messages
prerequisite constraints:	conditions to start a session (e.g. at least two people)
simultaneity constraints:	(restricted) access to a session
	messages in same order in every window
group decision:	agreement on communication channel (if the application doesn't use a server)

Multi-user drawing program:

transfer:	remote communication
	multi-cast for status updates
shared resource:	server
	locks to lock areas
prerequisite constraints:	register required to enter program
	lock only possible when area is unlocked
	unlock only possible when area is locked by unlocking user
simultaneity constraints:	no more than one user can use a drawing area at the same time
	deadlock prevention

4.2.3 Categorization of coordination solutions

The coordination solutions presented in the previous paragraph are application specific. We have categorized these solutions in a set of more general solutions, so that every solution should be reusable in more than one application. The result of this categorization is shown in table 1.

The row headings are the names of the sample applications, the column headings are the names of the coordination solutions. An $\surd$ indicates that the application uses this particular coordination solution. A coordination solution should be reusable for every $\surd$ in a column.

Draw	Chat	Game Server	Library	ATM	
✓	✓	✓	✓	✓	remote communication (transfer)
		✓		✓	request redistribution (transfer)
		✓			one-way connection (transfer)
		✓			two-way connection (transfer)
✓	✓				multi-cast connection (transfer)
	✓		✓	✓	database access (shared resource)
✓		✓	✓	✓	service access (shared resource)
✓	✓		✓	✓	infocheck before action (prereq)
	✓	✓			wait for condition before action (prereq)
	✓		✓	✓	access policy to shared resource (simult)
	✓				restricted access to service (simult)
✓					real-time constraints
		✓			group decision

Chapter 5

Communication

This chapter describes the communication part of the coordination framework. The goal of this part was to build coordination components to solve information transfer dependencies in open systems.

The coordination components are based on two basic communication mechanisms the Java language provides. First Java provides wrappers for socket communication. With these, socket connections can be set up and data can be transmitted over these sockets using streams. The second mechanism we used, is Remote Method Invocation (RMI)[1]. With RMI it is possible to do method calls on objects that are running on other Virtual Machines.

5.1 Stream based socket connections

The first mechanism we describe is string-based. These strings are sent over a TCP/IP network using sockets and streams.

First we have built abstractions to construct connections (see figure 5.1[2]). On the client side we have a `SocketConnection` that wraps the connection once it is instantiated. It provides a clear interface to write to and read lines from the connection. The establishment of the connection is encapsulated by the `Connector`. On the server side we have an `Acceptor`, which continuously waits for `Connector`s that try to connect. When the `Acceptor` accepts a connection it passes it on to a specialization of the `ConnectionMgr`. This manager determines what has to be done with the incoming connections.

With these abstractions it is possible to build communication layers for distributed applications. For instance configurations with a central server (see figure 5.2) or a ring based structure (see figure 5.3).

We built some sample applications using a central server. We first show a network game, and afterwards we show with a multi-chat facility, how we can reuse and configure the different communication abstractions.

In these central server based applications we put all the application dependent stuff (the "computational part") at the client side. On top of that we put a communication layer that takes care of the transfer of information between the clients.

[1]We tested the Pre-beta version of this feature for Java version 1.0.2. Now it is part of the newest version of the core Java language (version 1.1).

[2]For a notation guide, see appendix C.

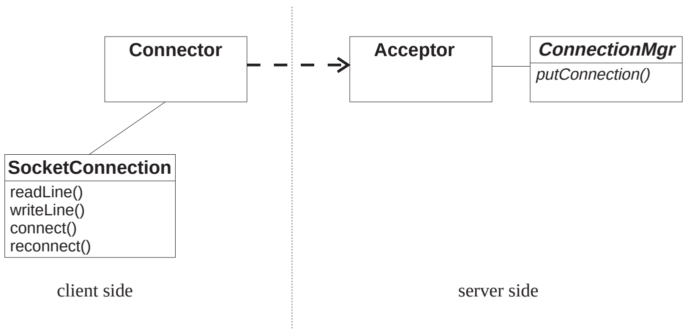


Figure 5.1: Connection abstractions

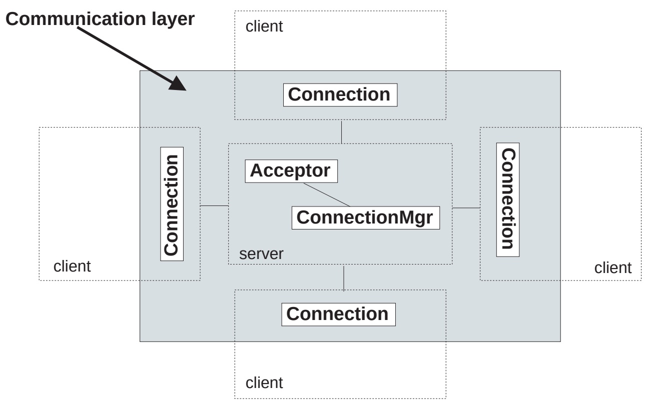


Figure 5.2: Communication with a central server

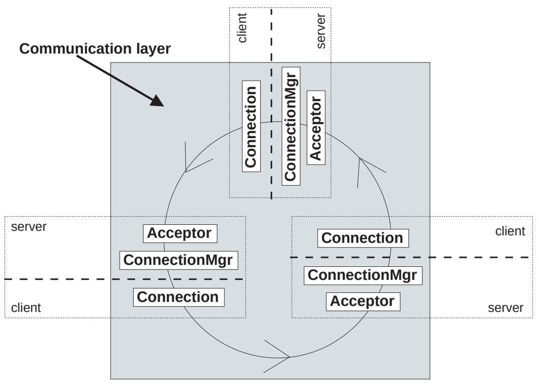


Figure 5.3: Communication with a ring based structure

The first example shows a simple one-to-one game using a central server (see figure 5.4). Clients connect to the game server and this server groups two incoming connections together.

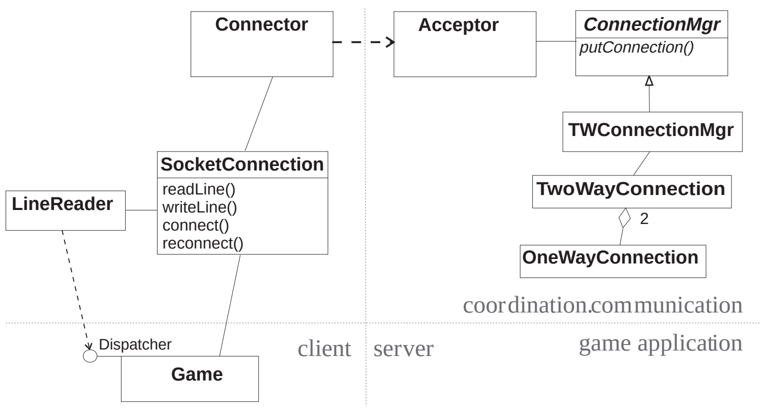


Figure 5.4: Network game example

We see some new coordination abstractions. On the client side we have the `LineReader`, which reads lines from the connection whenever a line is available, and passes this line on to an application part that implements a `Dispatcher` interface. On the server side we have the `TwoWayConnection-Mgr`, which connects two incoming connections via a `TwoWayConnection`, which uses two `OneWayConnection`
Once the connection between the two clients has been established, the connection between the two clients looks like in figure 5.5.

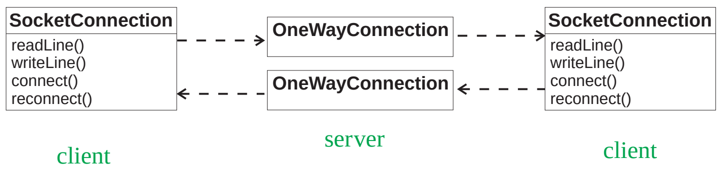


Figure 5.5: Two clients connected via two `OneWayConnections`

5.1.1 Reusability and flexibility

The above system is built as a reusable and flexible layer for asynchronous string based communication over a TCP/IP network.

We show the *reusability* by viewing another application using this communication layer. In figure 5.6 we clearly see what is reused. The client side of the communication is exactly the same with the black-boxes `SocketConnection` and `LineReader`. On the server side we reused the `Acceptor`, but we have another connection manager, namely the `MulticastConnectionMgr`. This connection manager uses the black box `MulticastConnection` to connect the clients so, that every incoming line from one of these clients is multi-cast to all connected clients.

With the connection abstractions at the server side: the `OneWayConnection`, the `TwoWay-Connection` and the `MulticastConnection`, we have a set of black-boxes that can be com-

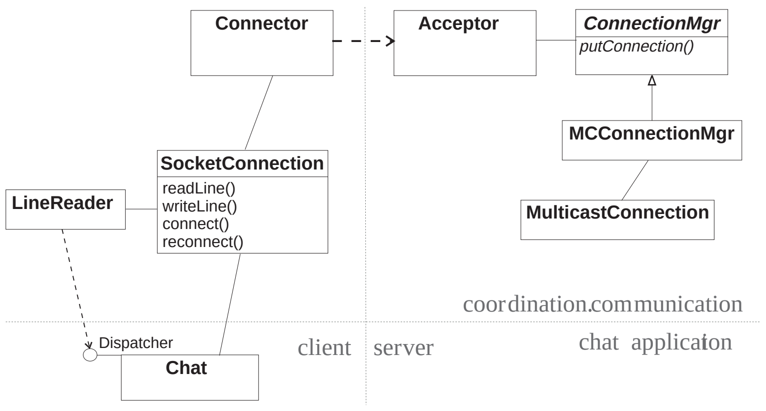


Figure 5.6: Multi-user chat application

bined and configured to set up different kinds of connections. Examples are the two connection managers and the `TwoWayConnection` shown, which is built using two `OneWayConnections`.

We gained *flexibility* by

- parameterizing the application by a connection. If we want to change the connection we can replace it (currently only at startup time) by another one with the same interface. In this way the actual application doesn't have to know which communication mechanism is used and to whom it is connected. This is specified during the configuration of the system. A configuration routine could look like this:

```
host = "server.somewhere.ch";
port = 6789;

// set up connection to host host and port port
conn = new SocketConnection();
conn.connect(host,port);

// set up game
game = new Battleship();

// set up linereader that reads from conn and puts its
// lines to game.
linereader = new LineReader(conn, game);
```

- The `Acceptor` is black box, and configurable, namely parameterizable by a `Connec- tionMgr`. This provides a clear decoupling of connection establishment and connection handling. Flexibility is reflected by the fact that both parts are now independently changeable. Again, we did no experiments with changing configurations at run-time.

5.1.2 Comparison with Java ACE

The Adaptive Communication Environment[35] of Doug Schmidt implements a set of design patterns for concurrent event-driven communication software. It provides abstractions for connection setup and connection handling. Comparable to our communication work are the Connector[36] and Acceptor[34] pattern. In these patterns connection establishment and connection handling are decoupled to increase reusability and flexibility.

The structure for setting up connections is more or less the same, but our approach is less restrictive:

- In ACE a `Reactor` is used to demultiplex multiple events in a single thread of control. In our approach it is open and transparent if single or multi-threaded solutions are used. This implies a simpler basic structure: our `Acceptor`/`ConnectionMgr` pair focuses on the communication problem, whereas in ACE complexity is added telling how to manage the Acceptor and the event handlers linked to the connections (by using a so-called `Reactor`). In our approach it is transparent how the connection is handled and our `Acceptor` is just a black-box delivering connections that appear on a certain port, to a connection manager. The `Acceptor` is a more independent component with a clear task.
- In our approach it is open if for every connection a connection handler is created or not. In ACE as it is presented, a handler is created for every connection. This is not always necessary. If we look, for instance, at our multi-cast abstraction, we see that this is one handler that handles multiple connections.

We claim that our approach is more simple and clear, and less restrictive. Therefore it is a better basis for building black-box coordination components: Our abstractions are more independent and therefore easier to reuse in different applications. It is also easier to plug in different solutions for establishing and managing connections, and these solutions decide, either themselves or by parameterization, how to act.

5.2 Remote Method Invocation

Remote Method Invocation (RMI) is a RPC-like mechanism in an object-oriented environment. The main idea is to extend the Java object model to a distributed object model in as seamless a way as possible. The general structure is shown in figure 5.7. A remote object is represented by a stub object at the client side. The `java.rmi` layer takes care of all the communication needs to do method calls on these remote objects.

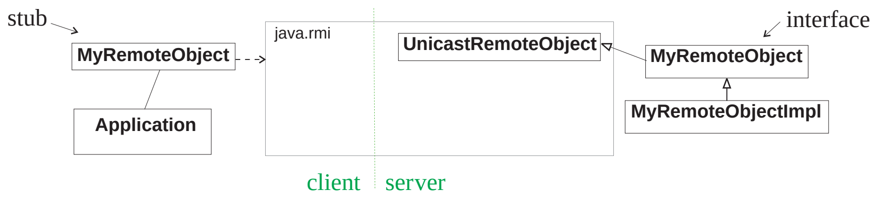


Figure 5.7: RMI

On the client side the use of RMI is fairly transparent. A method invocation on a remote object has the same syntax as a method invocation on a local object. The use of RMI is not completely transparent. There are two differences: first of all, before doing a method call on a remote object, there has to be a connection to this object. RMI uses a simple bootstrap name server to obtain remote objects on a given host. We wrote a simple abstraction that uses this server (thus making the use of this server transparent) and returns a remote object reference when given a host and the name of the object. The second difference is the fact that a remote method call can throw a (specialization of) a `java.rmi.RemoteException`. Every time a remote method is called this exception has to be caught or explicitly passed on to the next level of control (as every exception in Java). The possible failure of connections is a characteristic of distributed systems, so recovery should be taken care of. But the explicit exception mechanism of Java violates transparency of remoteness and the use of RMI as the communication mechanism.

On the server side, the use of RMI is far less transparent. A remote interface has to be made for every object that is intended for remote use. The implementation of this interface, i.e. the remote object itself, has to throw the earlier mentioned `RemoteException`. This violates transparency of local and remote use completely: even local calls on this object will have to catch this RMI-exception. The stubs for the remote objects have to be made using a special stub compiler and a Registry program has to be started at the server side. This is a kind of general remote object server, where the remote objects have to register themselves, before being remotely accessible.

If we look at the differences between RMI and sockets and streams, the main difference is that RMI is synchronous and socket/streams are asynchronous. This implies that they are not transparently interchangeable. We can think of a higher level mechanism that uses our socket stream abstractions to implement a higher level synchronous communication mechanism, for instance with send/receive pairs[3]. Most of the time this kind of connections is used, because when an (asynchronous) message is sent, we mostly need a notification of arrival, a notification that a request is carried out or a return value. We can also think of interchangeability with a CORBA connection. All these connection abstractions could probably have the same interface and would then be transparently interchangeable. Components that use these connections could then also be transparently used in different communication settings.

5.2.1 Conclusion

We looked at RMI as a communication mechanism in a component-oriented environment. We saw that in some ways RMI is a nice extension to normal method calls, and in some ways it isn't simple and transparent at all.

From the client point of view, RMI is a nice feature: it is almost transparent that remote objects are called, except that, before accessing a remote object, a connection has to be established. For the establishing of connections we wrote our own abstraction, that returns a reference to the remote object, when given the address and name of the remote object. Another point is that network failure exceptions have to be handled explicitly. This may be a good mechanism for building robust distributed applications, but it violates transparency principles.

From the server point of view, RMI is not transparent at all. The obligatory interface and exceptions, and the separate stub compilation make the use of RMI circumstantial and non-transparent. RMI could be more transparent, if every local object could be used remotely by defining a remote interface or by registering somewhere that an object is allowed to be remotely accessible.

[3]RMI itself is of course also a higher level synchronous communication mechanism based on socket connections

Chapter 6

Synchronization

Synchronization is, in addition to communication, a basic mechanism for building coordination abstractions. There are low level mechanisms like semaphores, mutexes and locks, but we can also think of complete coordination solutions for prerequisite or simultaneity dependencies.

In the following sections we will first discuss the basic synchronization mechanisms that Java provides (section 6.1). After that we will present some simple synchronization abstractions we built in Java (section 6.2). And in section 6.3 we present a distributed synchronization solution for locking objects over a network.

6.1 Synchronization in Java

Java provides a set of classes and constructs for concurrent programming. Concurrency is supported via so-called threads. Activities can be started in different threads of control, which causes the activities to run quasi-asynchronously. Execution of multiple threads can be controlled using `synchronized` constructs. These constructs ensure that only one thread at a time will enter a `synchronized` section in an object. Java takes care of this synchronization with an underlying mechanism based on the monitor and condition variable scheme developed by C.A.R. Hoare[13]. There are also a set of methods defined in `java.lang.Object` for managing threads. The most important ones are `wait()`, `notify()` and `notifyAll()`, respectively to put a thread in a wait state and to notify one or all waiting threads. The synchronization constructs provided by Java are very basic and sometimes somewhat crude. The thread scheduler of the Java Virtual Machine, for instance, is not fair: one is never 100% sure if a waiting thread will get a chance to run again. When a developer wants to be totally sure about a certain scheduling policy, he will have to explicitly take care of it himself. More information on Java concurrency mechanisms can be found in Doug Lea's book about concurrent programming[20].

We use the basic Java constructs to build more elaborated forms of synchronization. In the next section we show some basic synchronization abstractions. In section 6.3 we use these abstractions to build a distributed locking mechanism.

6.2 Simple coordination abstractions

In this section we give examples of simple coordination abstractions that encapsulate synchronization solutions. We start with a standard problem: the basic producer/consumer problem. This problem is, in coordination terms, a prerequisite problem: a consumer can not consume if there is not already something produced, and a producer can not produce something if the consumer didn't consume the earlier produced item. We need therefore to synchronize the production and consumption of the items.

A standard solution to this problem is the use of a one-slot buffer (see figure 6.1). The buffer provides a `put()` and a `get()` and takes care of the synchronization between these methods internally. This synchronization is fully transparent to the producer and the consumer: a call on the buffer which cannot be carried out due to synchronization constraints is blocked by the buffer until the synchronization constraint is satisfied (e.g. a consumer blocks until there's a new item available in the buffer). We don't have to limit ourselves to one-slot buffers. If we need some kind of buffering, for instance, to keep the producer from waiting if the consumer didn't yet consume an already produced item, we can take a many-slot buffer. And if there are more producers and consumers of the same kind of items, they can make use of the same buffer.

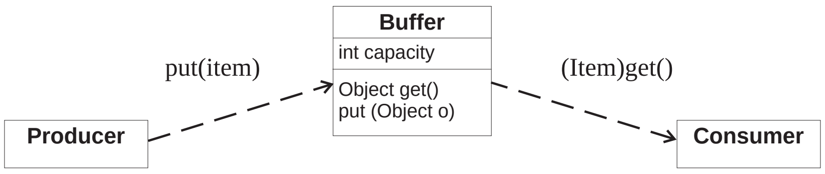


Figure 6.1: Buffer as synchronization solution between producer and consumer

This is a beautiful example of a synchronization component: the buffer encapsulates the synchronization between the producer and the consumer. If it can handle all kinds of items, it is reusable every time this kind of synchronization is needed. And it makes the architecture of an application more explicit by showing that there is an object taking care of the flow between producer(s) and consumer(s).

A slightly adapted version of the one-slot buffer is a future[3] (see figure 6.2). This is an abstraction that takes care of return values in an asynchronous environment. A method that will produce a result that is needed by the caller or somebody else, will return an object that will eventually hold the result value. Calls on this object for the result will be blocked until the result is available. The enforced synchronization is a write once/read many solution.

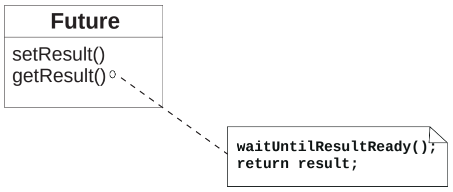


Figure 6.2: Future

The last abstraction we discuss is a lock[1]. Although Java offers its own locking mechanism via the `synchronized` construct, we may need more flexible and explicitly apparent locks. With the native Java mechanism it is, for instance, not possible to time out waiting for a lock. And we may need to explicitly reference a set of locks in order to implement a deadlock detection algorithm.

[1]For this part we extensively used the section about locks from the Concurrent Programming book of Doug Lea[20].

A `Lock` object will normally consist of an interface with methods for acquiring and releasing locks. Flexibility is increased when we add a separate method for checking the lock before entering a protected section. This makes it possible to have different activities sharing a key, so enabling group access to protected code. Flexibility is also increased by adding parameters to toggle the use of wait loops with or without time-outs, in order to be able to block requests in different ways until a lock is available.

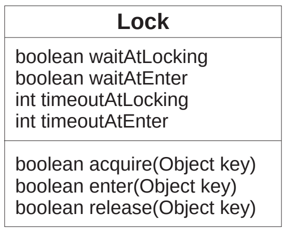


Figure 6.3: Lock component with methods to lock, unlock and check the lock and parameters to adjust its behaviour

By constructing a lock in this way we create a generic lock component with parameterizable properties. We make locks apparent in a design, supporting explicitness of architecture. And we are able to use this lock component in different settings depending on how it is configured. The lock component is shown in figure 6.3.

6.3 A distributed locking solution

We saw in section 6.2 a generic lock component. In this section we show how this lock is used to build a distributed locking solution. The problem we solved is a problem in the area of Computer Supported Cooperative Work: when multiple users have a common target on which they work concurrently, this target must be (partially) lockable by a user to avoid interference of different users. An example is an editor for editing a document by multiple users at the same time in a distributed setting. Lockable targets in this case could be documents or paragraphs of documents.

We will see in this section that the lock itself is just a small part of the total solution. A major part of the solution solves the problem of having consistent lock information available for every client. In figure 6.4 we show the structure of the solution. In the upper layer we used the socket communication described in section 5.1. In the middle layer we have the lock layer and the lower layer is the application dependent layer. In our editor example the application dependent layer should consist of editing and displaying routines.

According to the structure in figure 6.4 we first built an application that only uses a set of central locks and a user interface to lock the locks and to view the status of these locks. The solution at the client-side is shown in figure 6.5. The “lock layer”, the relevant layer in this section, consists of a `LockDispatcher` and a `ClientLockManager`. The `LockDispatcher` is a kind of filter: messages for the `ClientLockManager` are filtered out and sent to this manager; every other message is sent to the `Client`. The `ClientLockManager` manages the local lock information and takes care of the information exchange (using the communication layer) with the lock server. It provides a clear interface to the application: `register()`, `lock()` and `unlock()` and some methods to get status information about a lock. Objects that are associated with a lock are registered

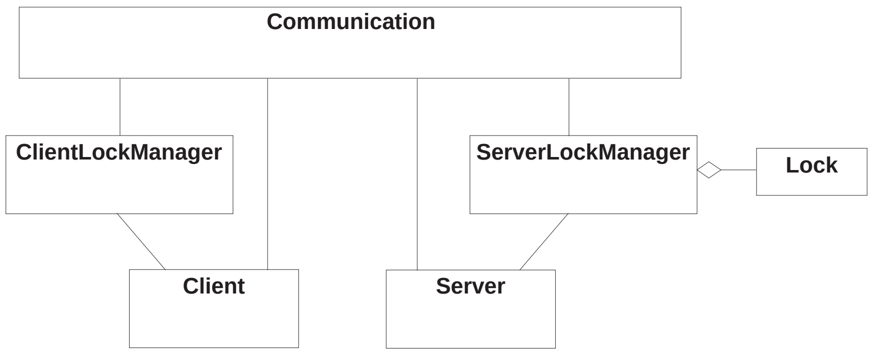


Figure 6.4: Overview of distributed lock solution

at the `ClientLockManager`, and then the application can lock and unlock[2] the object and ask information about the status of a lock.

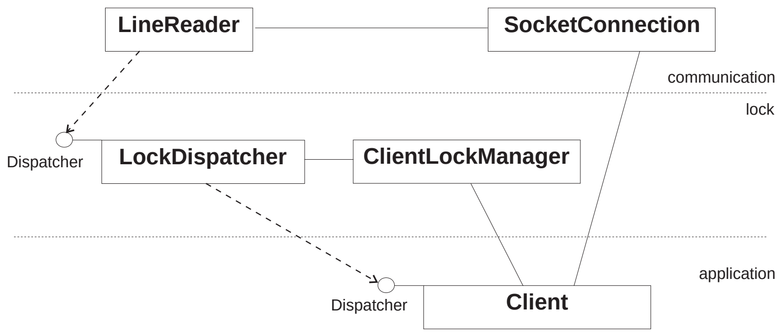


Figure 6.5: Lock solution - client side

On the server side we see the following structure (figure 6.6). The `LockManager` manages the locks, which are represented as `Lock` objects. Requests from clients for locking and releasing locks are dispatched by the manager, and status updates are multi-cast to the clients.

We reused the application independent part of this solution to build a simple drawing program, where multiple users can scribble in the same area, but only if they have locked it. This worked smoothly. The provided solution is limited in the fact that the number of locks is fixed during run-time. For a distributed editor where paragraphs are locked we can imagine that we need the number of locks to be flexible at run-time, because when editing a text it is very common that new paragraphs are created and existing ones are deleted.

On both the client and server side we have configuration routines that describe clearly how the components are parameterized and plugged together. A typical routine (which, in this case, sets up a client for the drawing program) looks as follows:

[2]In my sample application the lock and unlock operations are asynchronously sent to the lock-server. A request can fail, because the status information at the client wasn't up to date, for instance because a status update was on it's way to the client when the lock request was sent away. The status at the server is always the "right" state, and actions of clients that originate from inconsistent client states are ignored.

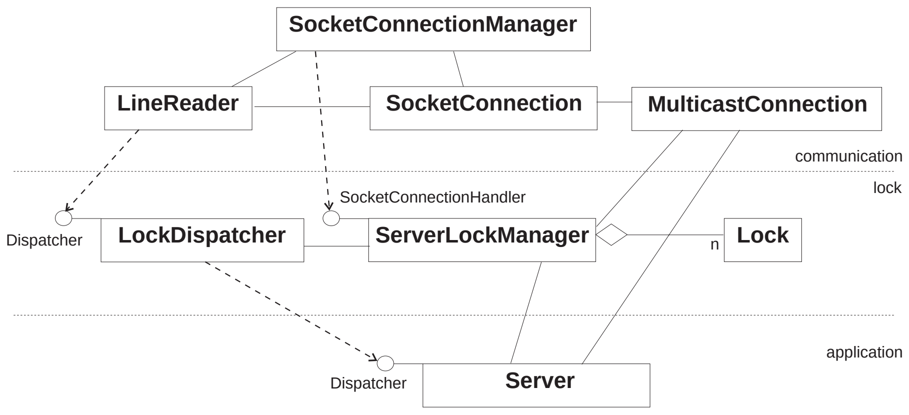


Figure 6.6: Lock solution - server side

```
// setup connection
conn = new SocketConnection(host,port);

// setup lock manager
lm = new ClientLockManager();
lm.conn = conn;
lm.init();

// setup first client
c1 = new ScribbleClient();
c1.conn = conn;
c1.lm = lm;
c1.x0 = 20;
c1.y0 = 60;
c1.init();

// setup clientlockdispatcher
ld = new ClientLockDispatcher(lm,c1);

// setup line reader
lr = new LineReader(conn,ld);
lr.sleeptime = 10;
```

In this routine the different components (see also figure 6.5) are instantiated and, if needed, some parameters are set. Behaviour of the application can be adapted by changing these parameters or by using another component with the same interface. An example of such a routine in a different setting is described in section 7.3.

Reusability and flexibility

The solution presented in this chapter provides an application dependent distributed locking solution. It has a clear interface to the rest of the application. Objects that should be associated with a lock[3] can register themselves and after that they can be locked or unlocked.

Distribution is transparent to the client. The proposed solution can be easily interchanged with a local locking solution with the same interface.

We also saw that the structure of the application is explicit as shown in a configuration routine.

[3]These objects mostly will be local representatives for objects that can be locked over a network

Chapter 7

Design of a reusable and pluggable Policy pattern

In this chapter we introduce a design for a reusable and pluggable policy pattern for the access of a shared resource in a concurrent environment. This pattern combines the Command and Strategy[1] pattern described by Gamma et al.[9]. It adds support for dispatching concurrent requests and flexibility is provided by making the policy context-free and, therefore, transparently changeable and usable in different applications. The Active Object pattern[19] has the same structure, but focuses more on how the policy should do its job and less on the transparency of these policies.

We start with an example to introduce the problem we want to solve with our design. Next, we state the requirements which our design has to cover and then we introduce our solution.

7.1 An example

We start with an example: the Automated Teller Machine. The basic structure of this example is described in figure 7.1.

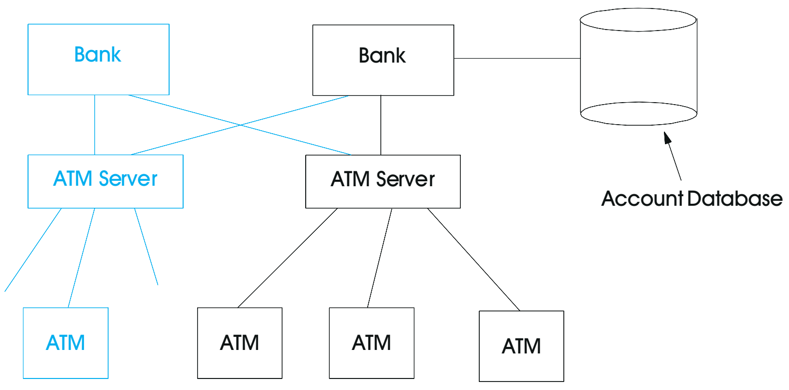


Figure 7.1: Automated teller machine system

The example describes a system to support a network of ATMs shared by a consortium of banks.

[1]also called: “Policy” pattern

Every bank has its own account database. Teller machines are connected to a server. This server redirects the ATM requests to the bank the request is supposed to go. The Bank serves the information requests from the ATMs.

For the purpose of this chapter we focus on the control of the requests from the bank to the account database. So actually this is an example of shared resource (database) access (see figure 7.2). The database offers a couple of (unsynchronized) actions, which the bank wants to access, possibly concurrently, to serve the ATM requests. Teller machines need to get information from the account database in order to check a client's account. They also need to update account information if they have transferred money to or from client accounts. To keep the database consistent we need an access policy to control the multiple requests.

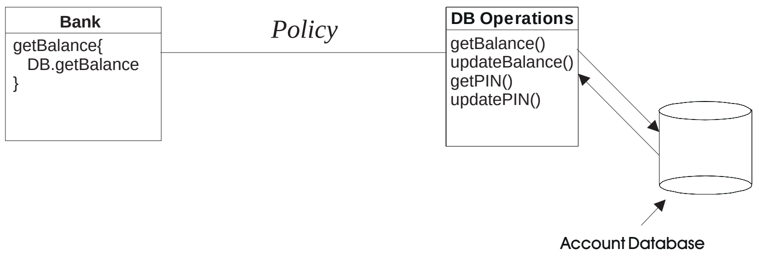


Figure 7.2: Database access in ATM system

7.2 Requirements

The main goal of our design is to implement the access policy to the database in a reusable and reconfigurable way. In terms of our component oriented approach, this means that we want to have a structure where we can plug in different policies, without having to change other parts of our solution. Together with the fact that we are designing for open systems, this leads to the following basic properties we want our solution to fulfill:

1. the whole solution should be able to dispatch concurrent requests.
2. the policy should be outside of Bank or DB Operations.
3. the policy should not only be outside but also as independent as possible from either Bank or DB Operations.

The third property is the most difficult to meet, because there are policies which need application dependent information (e.g. a readers/writers policy needs to know that `getBalance()` is a reader operation and `updateBalance()` a writer operation). We can distinguish three kinds of policies:

- policies that need no other information about the requests
- policies that need type information about the requests
- policies that need external information about the requests

We now go into more detail for each kind of policy.

Policies that need no information about the commands

This is the easiest kind of policy. They can do their job without any knowledge about the commands they have to dispatch. A typical example is a FIFO policy: no matter what commands come in, the policy just dispatches them in the order they come in.

Policies that need type information from the commands

This kind of policy depends on the ”nature” of the command: every command*type* has one or more properties which are needed by the policy. A typical example for this case is the readers/writer policy. The policy has to know if a command is a reader or a writer command. So this information must be made available in our solution.

Policies that need instance information from the commands

This kind of policy depends on information which can be different for each instance of a command. We can think of a priority policy, where, depending on the sender of the request, a command has a certain priority. Again, somehow, this information must be available to the policy.

7.3 Solution

The basic structure of our solution is shown in figure 7.3. We have an `Interface` of the solution to the rest of the application. Clients have to call this `Interface` to access the access solution. Of course, we have the resource itself. In between, we have a part which represents the control policy. To give the policy the ability to buffer the commands, change their order or execute them in parallel, we need an explicit representation of these commands. We do this according to the Command pattern of Gamma et al.[9].

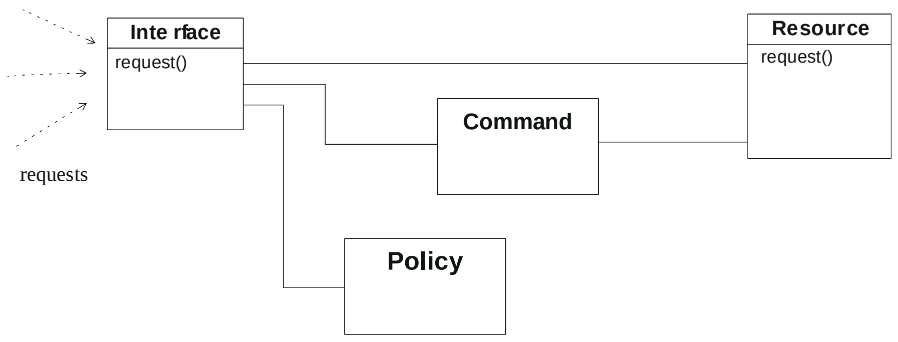


Figure 7.3: Policy overview

We first take a closer look at the Command design. Following, we look at the Policy part, before we give an overview of the total solution.

We set up the command part according to the Command pattern of Gamma et al.[9]. For the first kind of policy, the basic Command pattern suffices. For the second and third however, we need some additions (see figure 7.4).

For the second kind of policy we need to make type information available at run-time in order to be able to use this information at run-time, for instance to link this information to certain policy-dependent properties. We make this information available by providing a `CommandType` class to every subclass of the abstract `Command` class[2].

For the third kind of policy we need to make information available which can differ for every instance of a command. We do this by connecting a `Property` class to every `Command` class and the addition of a `SetProperty` and a `GetProperty` method to the `Command` class. What the exact information is, that subclasses of this class represent, is totally dependent on the application in which it is used. It could be that the name of the sender of the request is made available. Note, that the information, that is made available, shouldn't be information especially linked to a policy. As an example we take again the priority policy. In the `Property` object, we should make the sender of a request available. The linking of this sender with a priority is done later at the policy. We do this to keep the `Command` as independent of the policy as possible: the information is really a property of the `Command` and this information can be used by different policies.

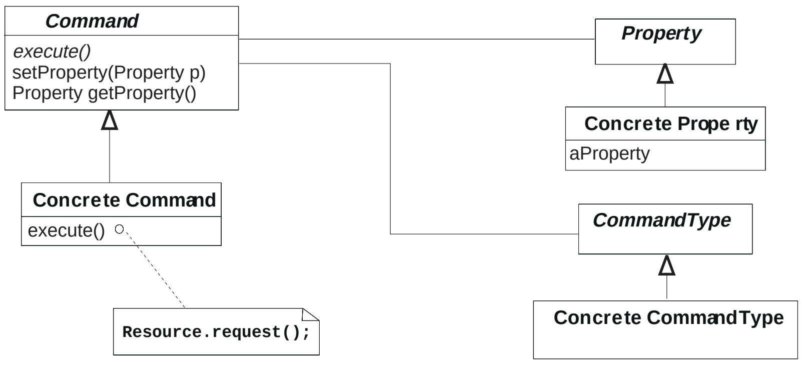


Figure 7.4: Command

The decision to take a `Property` class to hold information about commands has some advantages and disadvantages. The advantage is that the `Command` doesn't have to be aware of information that is made available about itself. It is also explicit in the architecture which information is made available and it is adaptable without having to change the `Command` class. The disadvantage is the overhead: the `Interface` may create `Property` classes and initialize information that may never be used.

In figure 7.4 we haven't taken into account that the `Command` classes may need a mechanism to return results. A common solution is the use of a "future" object, which takes care of the synchronization to a variable which will hold the result value in the future (see section 6.2 about synchronization abstractions). In figure 7.5 we show a situation where we used this abstraction. Commands that have return values are subclasses of `ReturnCommand`. This `ReturnCommand` uses the `Future` to provide a write-once-read-many mechanism[3].

We set up the policy part using the Strategy pattern[9]. According to this pattern we define a common interface for every policy we might need (see figure 7.6).

[2]We don't have to do this in Java, because in this language type information is available at run-time

[3]In this case the `ReturnCommand` doesn't actually return a `Future` object to the caller (as described in section 6.2), but it encapsulates the `Future`. The `Future`, however, implements the write-once-read-many policy and this synchronization is transparent even for the `ReturnCommand`.

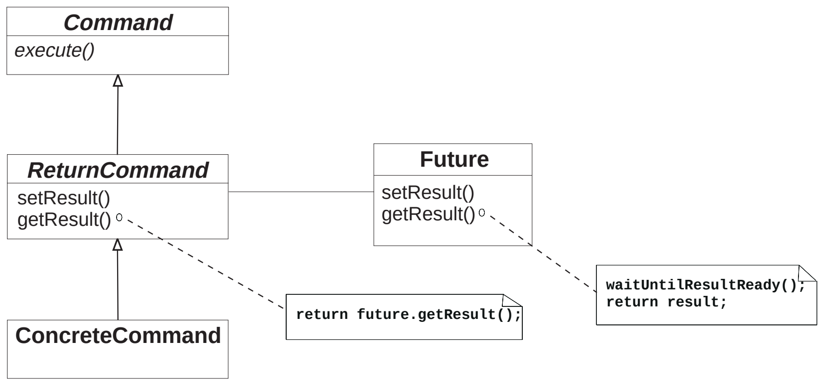


Figure 7.5: Future used for return values of explicit `Command` class

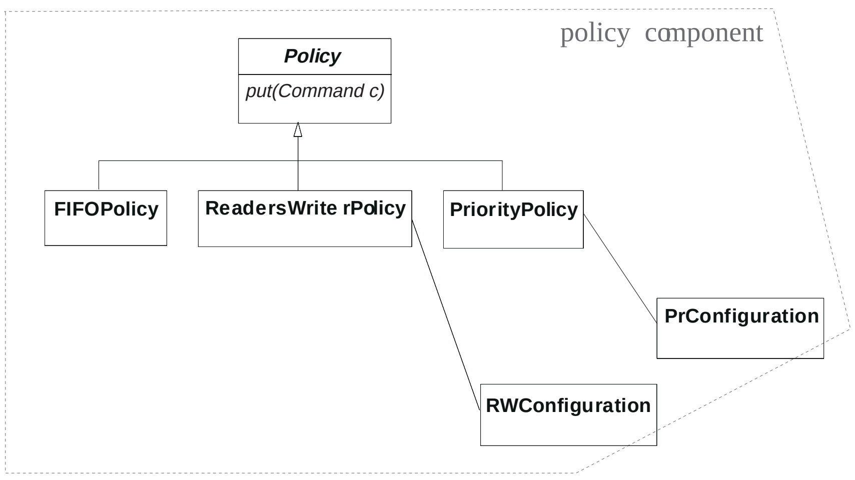
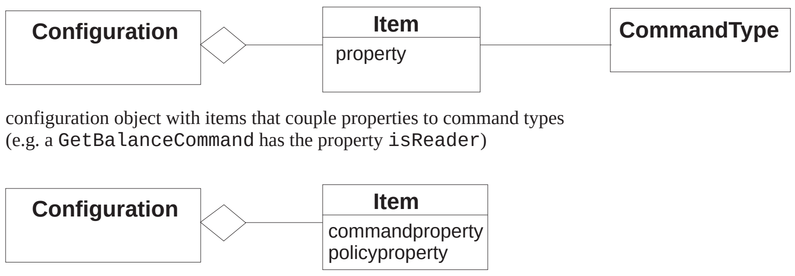


Figure 7.6: Policy

The hard part is again the fact that we have to deal with the application dependent information. The solution we propose is to represent this information explicitly in so-called ”configuration” objects. These configuration objects contain at run-time the information that is needed by the policies. So we may have, for example, a configuration object that contains of a link between command types and a property `isReader`. Another example is an object that links the names of possible request senders to a certain priority. We see here the difference between information that is made available in a `Property` object and that in a `Configuration` object. The former is general information, the latter contains the policy dependent information (see figure 7.7).



configuration object with items that couple properties to command types
(e.g. a `GetBalanceCommand` has the property `isReader`)

configuration object with items that couple command properties to policy properties
(e.g. `nameOfSender` to `priority`)

Figure 7.7: Configuration objects with different couplings

In figure 7.8 we see the total design for our shared resource access policy solution. The class `Interface` is the interface to the resource for the rest of the system. For every command which is invoked by an incoming request (1), a `Command` object is created (2). These commands are then given (3) to the policy which is connected (through parameterization) to the interface. This policy handles the commands, i.e. determines when and in which order the request can access the resource. If the command is allowed, it is executed (4).

We implemented this design in a toy teller machine application and a prototype library database application. We have used the same policies in both applications to show reusability. We also used different policies in the same application to show flexibility. The applications consist of the three shown policies, namely FIFO, readers/writer and priority. We managed to make these policies pluggable in the sense, that we can start the system with a parameter, which can be switched to get the wanted policy. According to the value of this parameter, the right policy is instantiated and, if necessary, the right configuration object. In the code below we see how the configuration of a system may look.

```
// initialization of policy according to chosen policy
if (chosenPolicy == FIFO) {
  pol = new FIFOPolicy();
}
else if (chosenPolicy == ReadersWriter) {
  // instantiate configuration object and add information
  conf = new RWConfObject;
  conf.add("GetBalanceCommand",isReader);
  conf.add("UpdateBalanceCommand",isWriter);
  // instantiate policy with configuration object as parameter
  pol = new ReadersWriterPolicy(conf);
```

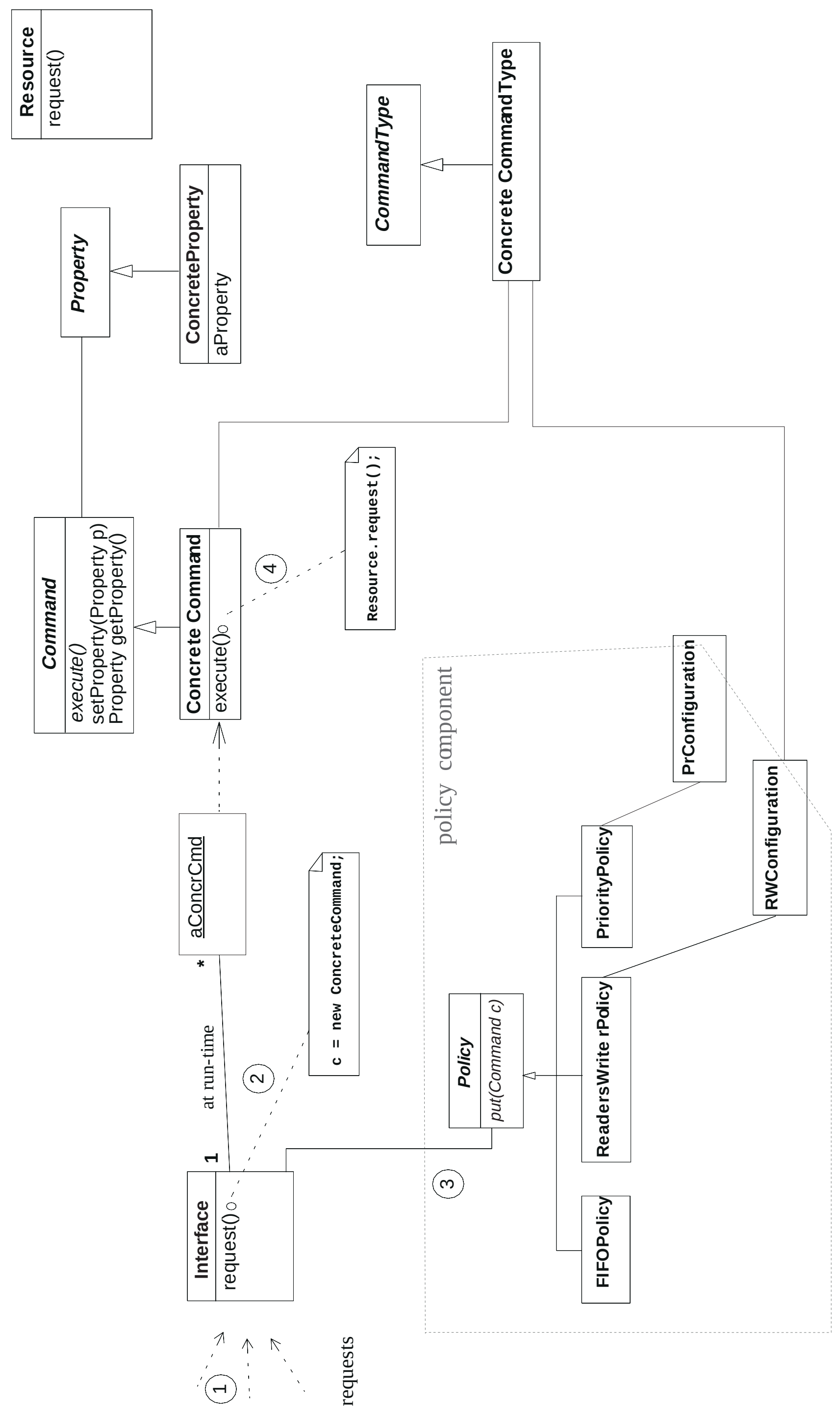


Figure 7.8: Policy component design

```
}
else if (chosenPolicy == Priority) {
  // instantiate configuration object and add information
  conf = new PrConfObject;
  conf.add(user1,10);
  conf.add(user2,15);
  // instantiate policy with configuration object as parameter
  pol = new PriorityPolicy(conf);
}
// start application with policy
new Application(pol);
```

For the FIFO policy we need no application dependent information, for the readers/writers policy we need the type information as described above and for the priority policy (which handles a `Command` with a certain priority according to the id of the sender of the request), we need to make the user id of the sender available via a `Property` object of a `Command`. Note, again, that this information is policy-independent: it can be used by other policies and doesn't effect the behaviour of policies that don't need this information (except maybe for some wasted memory).

7.4 Conclusions

We have presented an access solution for shared resources using a pluggable policy component. Depending on the policy we need in an application, we can transparently choose or change one of the available policies (*flexibility*). Or use a new one, if it conforms to the generic policy interface. It is also possible to use the same policies in different applications (*reusability*). We showed these features by implementing this design in different sample applications.

The policies are black boxes, which are transparently pluggable, if we have the right configuration objects available. These configuration objects contain the application dependent information which the policy needs in order to function. By using these configuration objects we can keep the policy black box, we have the application dependent information explicit in the design and therefore a clear configuration of the policy.

The information we covered in this way is related to the requests that come in: “Is a request a reader request?”, ”Which priority has the sender of a request?”. Up till now, we didn't take other application dependent information into account, like dynamic properties of the resource (for instance a check if a database is full before a write operation).

A drawback of this solution is the complicated pattern[4]: it will take some time to understand and to be able to use the pattern. Another disadvantage is the overhead. Objects are created for incoming requests. Information is made available in the `Property` object (while (parts of) this information may be never used by a policy at all). And every time a request comes in, the policy has to check its configuration objects. When performance is a critical issue this may be too much overhead. Optimal performance wasn't, however, a main goal in our design.

An open problem is the ability to switch policies at run-time. This is something we want, because some applications are long-lived of even permanent. Adjustments to these applications need to be

[4]This is, however, a direct consequence of the rather complex set of requirements

done while they are running. Run-time adaptability seems to be a straightforward extension to the presented design. We will have to add the ability to hold requests until a policy is switched, and there will have to be a mechanism to check if a policy is still busy (and if so, postpone the run-time switching).

Chapter 8

Request redistribution

In this chapter we look at a coordination problem in the area of transfer of information. We saw in chapter 5 some basic communication mechanisms. There we solved a basic problem of transfer dependencies: bringing a piece of data somewhere else. In this chapter we discuss a more complex problem: depending on some constraints, data can have different target locations. Our coordination solution will have to take care of the redirection of the data. As an example we take again a look at the ATM-example (see also section 7.1).

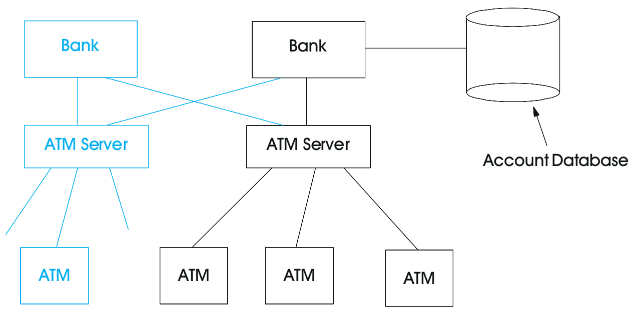


Figure 8.1: Automated teller machine system

In this example we have a request distribution problem at the ATM Server level: depending on the account involved in a transaction, a request has to be redirected to the bank that holds this account. In this case the redirection is triggered by the account number. This number determines to which bank (and thus to which location) the request has to be redirected.

The first observation we make is that the redirection problem consists of an application dependent part and an application independent part. The independent part consists of a generic name server that, depending on some identification, returns a connection, for instance a stub to a remote object. The application dependent part is the match of the account number to the generic identification. This basic structure is shown in figure 8.2.

The interesting part of this solution is the generic part, the name server (see figure 8.3). Inside this name server we have again a generic layer and a specialization for, in this case, the RMI communication mechanism (see section 5.2). The generic layer consists of the abstract `AddressConnector`, an `AddressList` and an abstract `Address` that forces concrete addresses to have an `id`. The RMI part consists of an `RMIAddressConnector`, which takes care of the RMI-specific implementation

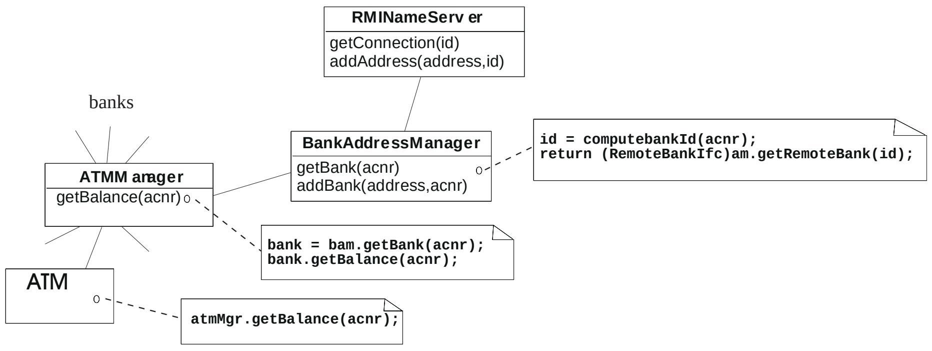


Figure 8.2: General request redistribution solution structure

details for connecting to a remote object , and a `RMIAddress` that holds the information needed as address for a remote object. The solution as a whole is black box with a clear interface.

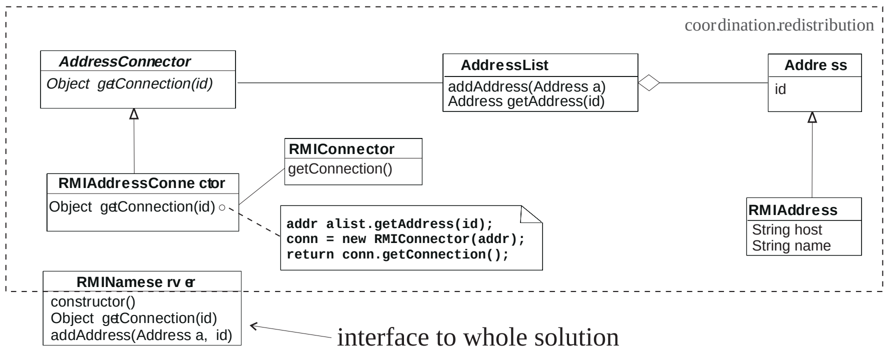
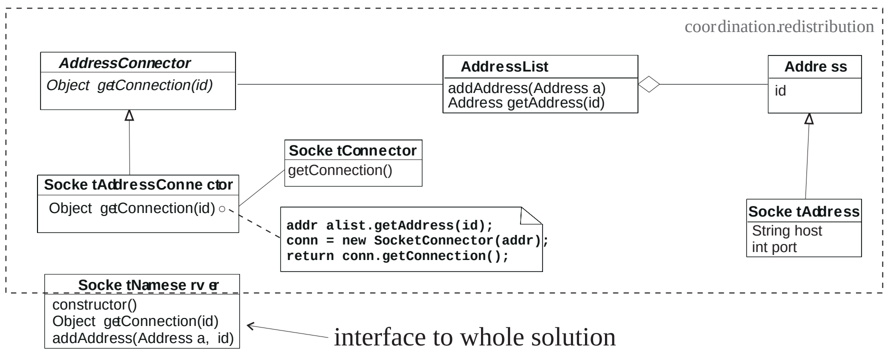


Figure 8.3: Generic part of the name server

Reusability and flexibility

It appeared hard to make a general solution to this problem. The reason for this is that a substantial part of the solution is application dependent. It consists of the process of choosing the right target location for a request. In the case of our banking system the target location is determined by mapping an account number to a bank identification. The nameserver, the generic part, is not much more than a set of tuples, consisting of two elements: an address and an identification. The nameserver returns a connection to an `Address` that is denoted by an `id`.

We see, only a part of the solution is generic. This part, however, is usable for different communication mechanisms. A solution that uses sockets (see figure 8.4), has the same interface, but returns a `SocketConnection` instead of a remote object.



```
addr alist.getAddress(id);
conn = new SocketConnector(addr);
return conn.getConnection();
```

Figure 8.4: Name server based on sockets

Part III

Conclusions

Chapter 9

Conclusions

In the research described in this thesis we have investigated software development for open distributed systems in order to make this development easier. Easier in the sense that software parts will be better reusable, more flexible and better maintainable. In particular we have investigated the coordination aspects of open distributed systems. To reach the goal of easier software development we have applied a component-oriented approach: generic coordination solutions are provided as generic architectures with black-box components.

We first developed a theoretical background about coordination abstractions in open distributed systems. The definition of coordination we used defines coordination as managing dependencies between activities. In open distributed systems, therefore, coordination is needed whenever dependencies exist between active entities. These dependencies are used to identify coordination problems in a set of sample applications that are representative for open distributed systems. We have implemented component solutions for a subset of these problems, thus building a prototype component framework.

The framework and sample applications have been programmed using the concurrent object-oriented programming language Java. This language is particularly well-suited to modeling software agents and components in a distributed setting. It provides low-level communication and synchronization abstractions that we have extensively used to build more elaborate forms of coordination[1].

We have to emphasize the fact that our work is a first attempt in building higher-level coordination abstractions. The presented results are, therefore, preliminary: many concepts need further investigation.

We first present a summary of the results of the different parts of our prototype framework. Our solutions range from basic communication abstractions (which are needed to build distributed systems in the first place) to more complex solutions like an access solution for shared resources and a distributed locking system. After that we will describe our general observations concerning the results of our approach.

[1]For a more extensive description of Java and its properties in the field of open distributed systems and coordination, see appendix B.

9.1 Results for framework parts

Communication

We provided communication solutions based on two mechanisms provided by the Java programming language. The first mechanism, based on sockets and streams, appeared to be a good mechanism to build generic communication abstractions. We built a set of communication components that were easily reusable and adaptable to a range of basic communication problems. We reused most our abstractions in four of our six sample applications[2]. The other mechanism, Remote Method Invocation (RMI), is a higher level mechanism that is used to do method calls on remote objects (i.e. objects that run on other Java Virtual Machines[3]). The fact that it is a higher level mechanism implies that we didn't have to care about how connections are set up, about providing synchronous communication over a network and about conversion from network communication to ordinary method calls. We found RMI a nice mechanism from the client point of view. For a client it is easy to use and fairly transparent: once the connection to a remote object is established it is accessible just like a normal object. From the server point of view RMI it is not easy to provide methods that are remotely accessible. Remoteness of an object is not transparent, which violates separation of concerns and reusability of the same components in both distributed and non-distributed settings.

Synchronization

We showed that it is possible to build some simple synchronization abstractions (a buffer, a future and a lock). We also built a distributed synchronization solution (the distributed locking solution). The biggest problem to solve in this distributed problem appeared to be to keep the integrity of the locks throughout the network, i.e. not more than one process can hold a lock and can do actions on an item that is protected by this lock. We managed to solve this integrity problem, but still there can exist some (inevitable) temporary information inconsistencies between clients and the server. Non-admissible actions, however, are prevented. The provided solution is black-box: we can view it as a set of higher-level coordination components with a clear interface to the application that uses the distributed locks.

Access policy solution for shared resources

For access to shared resources (in our sample applications: a shared database) we built an access policy solution. Different policy components can be transparently interchanged and be used in different applications. The interesting point in this design is how we cope with the application dependent information that is needed by the policy. We encapsulate this information in configuration objects. By using these objects we make the application dependent information explicit in the design and we keep the policies black-box and generic. A disadvantage is the complexity of the pattern.

Request redistribution

It appeared hard to develop a generic solution for the request redistribution problem. The main reason is that a substantial part of the solution (i.e. the determination of the target location) is application dependent. This reduces the generic part of the solution to an abstraction that returns a connection depending on some identification.

[2]see appendix A.

[3]see also appendix B.

9.2 General Conclusions

The solutions in the different areas show that it is possible to use a component-oriented approach to coordination problems in open distributed systems. Out of our experiences with the solutions in the different areas we can make some interesting, but preliminary, observations. Using a component-oriented approach we show that in many cases we gain

- *reusability*: our framework provides generic solutions for a set of coordination problems. Most of these solutions we use in more than one sample application (see appendix A). The possibility of reuse appears to be dependent on the application dependent information the generic abstraction needs to function. We will deal with this problem in more detail below.
- *flexibility*: most of the architectures, provided by our framework, can be easily adapted to provide other functionality by interchanging components with the same interface or by changing parameters to components. Evolution is addressed, because some of the flexibility is provided with future requirement changes in mind. An example is the policy pattern: when, in the future, a new policy is needed due to changed demands on the system, it can be transparently interchanged with the old one, as long as it has the same interface.
- *explicitness of architecture*: the designs provide a clear picture of what happens where in an application. In our applications this is shown in the configuration routines. In these routines all the components of a structure are instantiated and, if needed, parameterized, thus providing a clear description of the current configuration of a system.

The main problem we encountered in building generic components , is the application dependent information that a generic solution may need to be able to function. We found two different forms of information need in our experiments:

- We showed abstractions that need no or little information. The information that is needed, is directly parameterized in the solution. An example is a connection abstraction that is parameterized by a host name and a port number. These abstractions appeared to be easily reusable in different applications.
- In the policy pattern there is a more dynamic information need: behaviour of the application is dependent on the properties of incoming requests that can be different for every new request. We make the required application dependent information available via so-called *configuration objects*. By using these objects we make the application dependent information explicit in the design and we keep the policies black-box and generic.

Another observation we make is that it can be hard to develop generic coordination solutions, because (parts of) problems not only require application dependent information, but may as well require substantial application dependent computational parts. An example is the mapping from account number to bank id in the request redistribution solution.

Java

We found that Java, the implementation language of our prototype, has both advantages and disadvantages. A major advantage of the language is the built-in support for network communication and multi-threading. It appeared easy to use and useful for building coordination components. The Java thread scheduler, however, is *not* fair (see, for instance, [20]): one is never completely sure if a waiting thread will get a chance to run again. A major disadvantage of the language is the lack of genericity: the language doesn't provide a type parameterization mechanism like templates in C++. A more extensive discussion of advantages and disadvantages of Java can be found in [15]. Finally, we address the problems we had with the Java Remote Method Invocation mechanism. Based on this experience we can make the general observation that if (high-level) mechanisms violate component requirements (like encapsulation, etc), it can make them unsuitable for building components.

Further research

First of all, this project has been, as far as we know, a first attempt in developing a coordination component framework using a common object-oriented programming language. As a consequence our results are preliminary and further work needs to be done to generalize the concepts. Another consequence is that a part of the work has been the development of basic coordination abstractions, like communication. These communication solutions provide us with the means to build distributed systems in the first place and distributed coordination solutions in the second. We also built some solutions to more complex coordination problems, but a lot of work still can be done in providing solutions to other coordination problems.

We want to particularly mention the following areas where further research is required:

Application dependent information: it may be interesting to further investigate the information need of generic solutions to refine the categorization that we have presented above.

Distributed coordination solutions: Our lock server was a first attempt in building distributed coordination solutions, but more solutions are needed to come to more general observations.

Non-centralized coordination solutions: The communication and lock solutions we provided are all coordinated centrally. For reasons of fault tolerance this is not always desirable.

Run-time adaptability of systems: We provided some flexible solutions, but these are only adaptable at design-time. Systems, however, cannot always be stopped to change them. Therefore this extra level of flexibility is needed. One can think, for instance, in the direction of persistent configuration descriptions that can be adapted at run-time and newly interpreted to change the configuration. Most likely mechanisms like dynamic class-loading[4] may be useful.

[4] A feature of Java to load new or changed classes into an already running system, see appendix B.

Bibliography

[1] G. Bracha, “The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance”, Ph.D. thesis, Dept. of Computer Science, University of Utah, March 1992.

[2] N. Carriero and D. Gelernter, *How to write parallel programs*, The MIT Press, 1990.

[3] H. Baker and C. Hewitt, “The Incremental Garbage Collection of Processes”, *Proc. of Conference on AI and Programming Languages*, Rochester, NY, 1977.

[4] J.-P. Banâtre and D. Le Métayer, “Gamma and the Chemical Reaction Model,” *Proceedings of the Coordination’95 Workshop*, IC Press, Londres, 1995.

[5] N. Carriero and D. Gelernter, “Linda in Context”, *Communications of the ACM*, vol. 32, no. 4, April 1989, pp. 444-458.

[6] J.C. Cruz, S. Tichelaar and O. Nierstrasz, “A Coordination Component Framework for Open Systems”, submitted to COORDINATION’97.

[7] J. Ferber, *Les Systemes-Multiagents: Vers une intelligence collective*, Intereditions, 1995.

[8] D. Flanagan, *Java in a Nutshell*, O’Reilly & Associates, February 1996.

[9] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns*, Addison Wesley, Reading, MA, 1995.

[10] D. Garlan and M. Shaw, “An introduction to Software Architecture”, *Advances in Software Engineering and Knowledge Engineering, Volume I*, V. Ambriola and G. Tortora (eds), World Scientific Publishing Company, New Jersey, 1993.

[11] D. Gelernter and N. Carriero, “Coordination Languages and their significance”, *Communications of the ACM*, vol. 35, no. 2, February 1992.

[12] J. Gosling and H. McGilton, *The Java Language Environment*, Sun Microsystems, Inc., May 1995.

[13] C.A.R. Hoare, “Monitors: An Operating System Structuring Concept”, *Communications of the ACM*, vol. 17, no. 10, Oct. 1974, pp. 549-557.

[14] ISO/IEC JTC1/SC21/WG7, “Reference Model of Open Distributed Processing”, Draft International Standard ISO/IEC 10746-1 to 10746-4, Draft ITU-T Recommendation X.901 to X.904, May 1995.

[15] P. Jain and D. C. Schmidt, “Experiences Converting a C++ Communication Framework to Java”, *C++ Report*, SIGS, Vol. 9, No. 1, January, 1997.

[16] R.E. Johnson and B. Foote, “Designing reusable classes”, *Journal of OO Programming*, June/July 1988.

[17] T. Kielmann, “Designing a coordination model for open systems”, *Proceedings COORDINATION’96*, P. Ciancarini and C. Hanking (ed.), LNCS 1061, Springer-Verlag, 1996.

[18] J. Kramer, J. Magee and A. Finkelstein, “A Constructive Approach to the Design of Distributed Systems”, *Proc. of the 10th Int. Conf. on Distributed Computing Systems*, Paris, pp. 580-587.

[19] G. Lavender and D.C. Schmidt, “Active Object: an Object Behavioral Pattern for Concurrent Programming”, *Proceedings of the 2nd PLoP*, September 1995.

[20] D. Lea, *Concurrent Programming in Java: Design Principles and Patterns*, Addison-Wesley, 1996.

[21] T.W. Malone and K. Crowston, “The interdisciplinary study of coordination”, *ACM Computing Surveys*, Vol. 26, No. 1, March 1994.

[22] J. Magee, N. Dulay and J. Kramer, “Structuring Parallel and Distributed Programs”, *Proceedings of the International Workshop on Configurable Distributed Systems*, London, March 1992.

[23] J. Magee, N. Dulay and J. Kramer, “Specifying Distributed Software Architectures”, *Proceedings ESEC’95*, LNCS 989, Springer-Verlag, 1995.

[24] C. McHale, “Synchronisation in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance”, Ph.D. Dissertation, Department of Computer Science, Trinity College, Dublin, 1994.

[25] M.D. McIlroy, “Mass Produced Software Components”, *Software Engineering*, P. Naur and B. Randell (ed.), NATO Science Committee, January 1969, pp. 138-150.

[26] T.D. Meijler and O. Nierstrasz, “Beyond Objects: Components”, *Cooperative Information Systems*, M. Papazoglou (Ed.), Academic Press, London, to appear.

[27] Microsoft Corporation, *Visual Basic Programmer’s Guide*, 1993.

[28] M.J. Miller, “Rebuilding Componentware”, *PC Magazine*, June 25 1996.

[29] H. Mintzberg, *The structuring of organizations*, Prentice Hall, 1979.

[30] O. Nierstrasz, “Composing Active Objects”, *Research Directions in Concurrent Object-Oriented Programming*, G. Agha, P. Wegner and A. Yonezawa (Ed.), MIT Press, 1993, pp. 151-171.

[31] X. Pacheco and S. Teixeira, *Delphi Developer’s Guide*, Borland Press/Sams Publishing.

[32] D.E. Perry and A.L. Wolf, “Foundations for the study of Software Architecture”, *ACM Software Engineering Notes*, vol. 17, no. 4.

[33] Rational Software Company, *UML Document Set*, Version 1.0, January 1997.

[34] D.C. Schmidt, “Acceptor – A Design Pattern for Passively Initializing Network Services”, *C++ Report*, SIGS, Vol. 7, No. 8, November/December 1995.

[35] D.C. Schmidt, “ACE: an Object-Oriented Framework for Developing Distributed Applications”, *Proceedings 6th USENIX C++ Technical Conference*, Cambridge Massachusetts, USENIX Association, April 1994.

[36] D.C. Schmidt, “Connector – a Design Pattern for Actively Initializing Network Services, *C++ Report*, SIGS, Vol. 8, No. 1, January 1996.

[37] M.Shaw, D.Garlan, R.Allen, D.Klein, J.Ockerbloom, C. Scott, M.Schumacher, “Candidate Model Problems in Software Architecture”, *Discussion draft 1.3 in circulation for development of community consensus*, November 1994.

[38] G. Tel, *Introduction to Distributed Algorithms*, Cambridge Press, 1994.

[39] D. Tsichritzis, “Object-Oriented Development for Open Systems”, *Proceedings IFIP’89*, North-Holland, San Fransisco, 1989, pp. 1033-1040.

[40] P. Wegner, “Dimensions of Object-Based Language Design”, *Proceedings OOPSLA‘87*, ACM SIGPLAN Notices, vol. 22, no. 12, December 1987, pp. 168-182.

Part IV

Appendices

Appendix A

Coordination abstractions in the sample applications

In table A.1 we show where the abstractions of our framework are used in the sample applications. Vertically the abstractions are shown and horizontally the sample applications. A $\surd$ means that the indicated abstraction is used to build the indicated sample application.

Draw	Lock	Chat	Game Server	Library	ATM			
✓	✓	✓	✓			SocketConnector	socket/stream	communication
✓	✓	✓	✓			SocketAcceptor		
✓	✓	✓	✓			SocketConnection		
✓	✓	✓	✓			SocketConnectionManager		
✓	✓	✓	✓			LineReader		
			✓			OneWayConnection		
			✓			TwoWayConnection		
✓	✓	✓				MultiCastConnection		
				✓	✓	RMIConnector	RMI	
				✓	✓	BoundedBuffer		synchronization
				✓	✓	Future		
✓	✓					Lock		
✓	✓					LockConnectionManager		
✓	✓					LockManager		
✓	✓					ClientLockManager		
				✓	✓	Command		policy
				✓	✓	FIFOPolicy		
				✓	✓	PriorityPolicy		
				✓	✓	ReadersWriterPolicy		
					✓	RMINameserver		redistribution

Table A.1: Overview of abstractions in sample applications

Appendix B

Java

We used the programming language Java[12] to build our sample applications and the prototype coordination framework. Java is a concurrent object-oriented language developed by Sun Microsystems. It was first developed as a programming language for embedded systems. The imposed requirements - small, reliable and architecture neutral - made the language particularly fit to do network programming. It became especially popular in parallel with the World Wide Web and the Internet in general, because it is easy to use Java to build (inter)active contents for WWW pages.

Java has some special properties in the area of open distributed systems and coordination. We discuss these properties after mentioning some general properties of the language[1].

B.1 Java in general

Java is a C++-like language, but it lacks several features that C++ has. The most important one is probably the lack of pointers, which is normally an error-prone aspect of programming languages. Object referencing and dereferencing is handled by the language itself. Another point is that Java offers automatic garbage collection: the memory that is used by objects that are not referenced anymore, is automatically freed by the run-time environment. This makes the language easy to use.

Java is an interpreted language. The code is compiled into an intermediate form, called “byte-code”. This byte-code can be transferred over a network and executed on every platform that implements a Java interpreter and Java run-time system (together called the Java Virtual Machine). This Virtual Machine runs in the same way on every platform, thus making Java platform independent.

Java is a secure language. It is not possible to directly access memory on a system. The Virtual Machine takes care of all the memory decisions. There are also other security checks made by the Virtual Machine, like run-time type checking of newly created classes.

B.2 Java and Components

Java offers the normal object-oriented mechanisms to construct and compose components. Classes, objects (at run-time), methods and packages (the Java module concept) as components; inheritance and instance connections as class and object composition mechanisms.

Java doesn’t have multiple inheritance. Instead it has so-called *interfaces*. A class in Java can extend one superclass and implement multiple interfaces. An interface is a kind of abstract class

[1]For this chapter we extensively used the book “Java in a Nutshell” by David Flanagan[8].

with only abstract (i.e. non-implemented) methods. An object can be accessed via every interface it implements. This mechanism enables an object to provide different views to different clients.

A disadvantage for component development is the fact that Java doesn't provide genericity (i.e. type parameterization) and parameterizing by functions.

B.3 Java and Open Distributed Systems

Java is designed to develop network programs (and thus distributed systems). It provides a networking package (`java.net`) that includes abstractions for network connectivity, like a `URL` class to access remote objects over the Internet and classes to set up reliable stream network connections using sockets.

Another mechanism provided by the Java language is Remote Method Invocation (RMI). With this mechanism it is possible to do method calls on objects that reside on other Virtual Machines[2].

Openness is supported by the architectural neutrality of the language and the portability of the byte-code. Code can be transported over networks and run on every platform that implements the Java Virtual Machine. The language also supports run-time adaptability of applications. Due to the interpreted nature of Java, it is possible to dynamically load new or adapted classes into a running system.

B.4 Java and Coordination

Coordination is about managing the interaction of multiple, possibly parallel, activities. In Java multiple parallel activities are supported by the multiple thread support and the synchronization primitives to control these threads. The `java.lang` package provides a `Thread` class that supports methods to start and stop a thread, and to check on the status of a thread. Activities can be started in different threads of control, which causes the activities to run quasi-asynchronously. Execution of multiple threads can be controlled using `synchronized` constructs. These constructs ensure that only one thread at a time will enter a `synchronized` section in an object. Java takes care of this synchronization with an underlying mechanism based on the monitor and condition variable scheme developed by C.A.R. Hoare[13]. The synchronization constructs provided by Java are very basic and sometimes somewhat crude. The thread scheduler of the Java Virtual Machine, for instance, is not fair: one is never 100% sure if a waiting thread will get a chance to run again. When a developer wants to be totally sure about a certain scheduling policy, he will have to explicitly take care of it himself. Together with the networking support the synchronization primitives can be used to coordinate distributed activities. More information on Java concurrency mechanisms can be found in Doug Lea's book about concurrent programming[20].

[2]For a discussion of RMI, see section 5.2

Appendix C

The Unified Modeling Language (UML)

We used the Unified Modeling Language (UML)[33] to draw the OO designs presented in this thesis. The UML is a language for specifying, constructing, visualizing, and documenting the artifacts of a software-intensive systems. It is built upon similar semantics and notation from Booch, OMT, OOSE, and other leading methods. Version 1.0 of the language is submitted to the Object Management Group (OMG) to be considered for adoption as a standard. In this appendix we only present a small subset of the UML. We only show the symbols we have used in this thesis. More information can be found in [33] which is online available on the World Wide Web at `http://www.rational.com/ot/uml/`.

In figure C.1 we show the different symbols we used in this thesis.

A class is drawn as a solid-outline rectangle with 3 compartments separated by horizontal lines. The top name compartment holds the class name; the middle list compartment holds a list of attributes; the bottom list compartment holds a list of operations (example: `Class1`). Either or both of the attribute and operation compartments may be suppressed. A separator line is not drawn for a missing compartment. If a compartment is suppressed, no inference can be drawn about the presence or absence of elements in it (examples: `Class2`, `Class3` and `Class4`). Strings for the names of abstract classes or the signatures of abstract operations are displayed in italics (example: `AbstractClass1`).

Interfaces are shown as classes (i.e. using full rectangles). There is, however, a shorthand notation: a small circle with the name of the interface (example: `Interface1` which is implemented by `Class2`). The circle may be attached to classes (or higher-level containers, such as packages that contain the classes) that support it by a solid line. This indicates that the class provides all of the operations in the interface type (and possibly more). The operations provided are not shown on the circle notation; to show the list of operations the full rectangle symbol is used. A class that requires the operations in the interface may be attached to the circle by a dashed arrow.

A binary association is drawn as a solid path connecting two class symbols (example: the line between `Class1` and `Class3`). A hollow diamond is attached to the end of the path to indicate aggregation. The diamond may not be attached to both ends of a line, but it need not be present at all. The diamond is attached to the class that is the aggregate (example: the line between `Class1` and `Class4`).

Generalization is shown as a solid-line path from the more specific element (such as a subclass) to the more general element (such as a superclass), with a large hollow triangle at the end of the

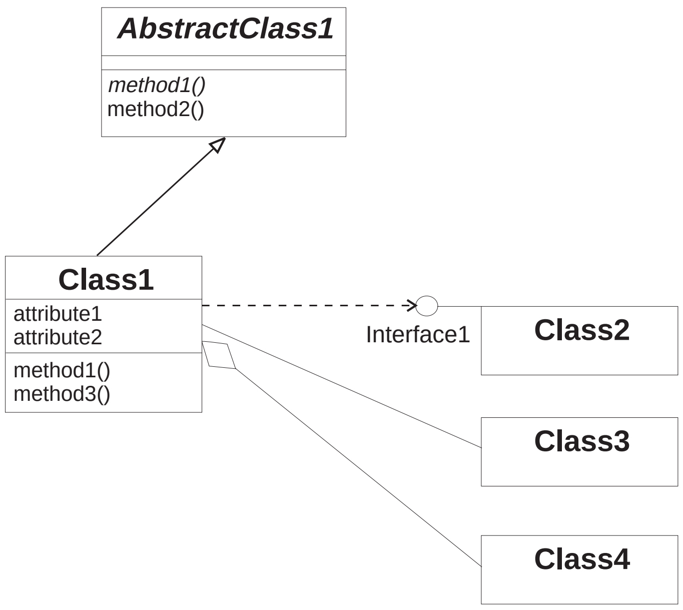


Figure C.1: Class structures according to the UML

path where it meets the more general element (example: the arrow between `AbstractClass1` and `Class1`).

The object notation is derived from the class notation by underlining instance-level elements (see figure C.2).

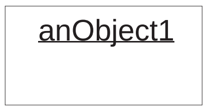


Figure C.2: An object according to the UML