Mobile Agents as a Distributed Application Architecture

Remco Slotboom

Master thesis for completion of a degree in Business Informatics at the Erasmus University Rotterdam.

Supervision by Dr. Ir. Jan van den Berg and Drs. Mark Polman.

Supported by Cambridge Technology Partners.

Contact information
Remco Slotboom
Ellemare 193
3085 JR Rotterdam,
The Netherlands.

Email: rslotb@ctp.com

Student number: 126801

Table of Contents

Preface

What does it take to create a master thesis? You take:

- about 30 packs of Earl Grey tea
- 300 litres of hot water
- a pile of agent literature
- a stack of other subject matter literature
- a computer

Add one human on overdrive, and mix for six months.

You get? One master thesis on agents. Mobile agents specifically, and how they can be used to create architectures for distributed computer systems. If anything, I hope this thesis will *not* contribute to the current hype on agents, but instead show some real, present time advantages of the concept.

As you can suspect, loads of hard work went into the creation of this document. In the next section, I will briefly describe how this document saw the light. After that, I want to extend my thanks to some people that were crucial to the completion of this thesis.

How this thesis came to be

While working as developer on a large-scale e-commerce application, I came into contact with an OO consultant who suggested ‘mobile agents as a new paradigm for distributed computing’ as a possible master thesis subject. Roughly parallel to that, one of my colleagues was very excited about aglets (IBM’s name for mobile agents) and kept insisting just how cool they were and what daunting possibilities they had. All this slumbered a bit for some time, because the breakneck schedule of the application under development allowed me to focus on little else.

Initial preparations for this thesis started somewhere at the beginning of 1999, when I opted to do the actual writing of the thesis in a reduced timeframe, but remaining employed. The basic idea was that if I could do some preparation (reading, research) while working, I would be able to write the thesis in three months full-time, with some of my own holiday time added.

‘Agents’ was actually a second, last resort option chosen when an earlier proposal was shot down for being ‘beaten to death’. Nevertheless, everybody loved the subject, and I started my initial explorations into the literature.

I ploughed trough a large pile of books and articles on agents, mobile agents and more general material on AI and distributed computing. From these, I gradually assembled the pieces of the puzzle, and gained new insights. Most of these articles presented different, even conflicting views of agents, which caused me to present, after thorough analysis, my own view of agents described in chapter 2.

During our previous development, I was confronted with the complexities of creating large scale distributed applications, and I gradually put two and two together, seeing how mobile agents could be used as a possible solution for many of the issues found in modelling and developing such applications. This is why I chose to focus my efforts on presenting mobile agents as a distributed application architecture; the title of this thesis.

My interest in OO modelling and the need for thorough analysis and design for such large scale applications led me to investigate how agents fitted into object oriented design and modelling, as presented in chapter 4.

Finally, theory is one thing, but I personally don’t believe something really works until I see it working. Also, building a prototype is one of the best ways a theory like the one I present can be validated and tested. An initial prototype proved that it really was surprisingly easy to implement agents in Java, and that it provides a very elegant programming model. In chapter 6 we show some ways agents and mobile agents can be implemented, and chapter 7 presents the design and implementation of a working prototype that is part of this thesis.

The prototype is a web based application based on mobile agents, called WebWatcher. The initial version allows you to create ‘smart’ bookmarks that will alert you when a page you indicated interest in is changed. It should be on line by now, at:

> http://webwatcher.remmie.com

Please, visit it and tell me what you think.

A word of thanks

I am very grateful to a lot of people for giving me the opportunity and supporting me. I want to thank the following people explicitly:

Jan van den Berg and Mark Polman for their relentless but ever fair comments on my ramblings, Richard Narine, Kristen Russell, and Clive Bourke for being killer PMs, Anees Qureshi as a great mentor, Albert Bartels for pushing me over the edge, Edward van der Jagt for being network guru, Richard Horner, Erik Rutten, PH de Wit and Jean-Paul de Vooght for being inspiring colleagues, Jacques Heesterbeek, Bill McCarthy, Onno de Cloe, Paul Piebinga and Eric Jansen for their support.

And of course a very special 'can't thank you enough' thanks to Indra Caerteling for always being there.

(Late night) inspiration came from:
Massive Attack, Prodigy, Terry Pratchett, Red Snapper, Junkie XL, Douglas Adams, Moby, J. Michael Straczynski, Kruder & DorfMeister, George Marinov (Stereoman), Tobias Wilton, TMF, Underworld, Orbital, Andreas Saag, Chemical Brothers, MTV, The Lego Group.

Remco Slotboom, April 2000.

CHAPTER 1 Introduction

Why are mobile agents so fascinating? Because we believe we can do things with mobile agents that are very hard to accomplish otherwise. This first chapter motivates why mobile agents are so interesting, and gives a quick overview of things to come. We start with an executive summary, which in a nutshell describes the main ideas presented in this thesis. You can use it as a first quick overview, or to decide whether this document is relevant for you or not. We continue with the motivations and goals of this document. The last section is an outline of the following chapters with some guidance to the reader.

1.1 Executive summary

This thesis investigates the possibilities of agents, and especially mobile agents, as a distributed application architecture.

Agents are software components that operate more or less on their own, or autonomously. As a software architecture, agents are a way to build autonomous computer systems that operate largely asynchronous, not depending on user interaction. Such systems can be delegated certain tasks, much like a real person can be delegated a task. There is a whole range of applications for this, from digital personal assistants to fully automated factory control systems.

Because being able to operate autonomously requires some degree of intelligence, agents are often ‘intelligent’ agents, employing some form of artificial intelligence. We describe several techniques that can be used to create intelligent agents.

The next step is mobile agents, which are agents that are able to move themselves from machine to machine. This represents a radical shift from more traditional distributed computing architectures. It allows us to freely move code around on a network, which in turn enables us to have a much more flexible architecture with a much better network performance. In addition, it can cater for systems that incorporate both high and low bandwidth connections and with processing power unevenly distributed around the network.

To support building systems based on (mobile) agents, we also show how such systems can be designed. We enhance the standard modelling language UML with a few additions that allow us to better model (mobile) agent based applications and even distributed applications in general.

To help implementation of (mobile) agent based system, we discuss what existing software can be used to implement it, and present several techniques that can be used to create agents and mobile agents.

Part of this thesis is the design and implementation of a prototype application that highlights the features of mobile agents.

WebWatcher URL

The prototype should be online by now, and can be found at:

http://webwatcher.remmie.com

We end with some final conclusions and future directions illustrating what could happen if we take the concept of mobile agents to the limit.

1.2 Motivation

Have you ever wished your computer truly do something for you, do something so you don't have to do it yourself, like check a web page to see if its updated, or automatically sort your email? That something is software agents can do. Now, have you ever wanted your computer to be a little more proactive? Maybe try to help you out when you are obviously stuck, or maybe tell you if there is something wrong, and help you fix it? Again, that's something that can be done with *software agents*. Software agents are a very useful model for building software that needs to act autonomous. An example could be an agent that guides you through a web site, or places bids in your name in an online auction. Some agents don't even have any interaction with a user, but do things like managing a system of valves in a refinery, or reporting changes in a robot design to another system.

Now, wouldn't it be great if the computer would do something for you even when it was turned off? Of course, this is physically impossible[1], but we could have your computer hand off the task to a computer that is permanently on, and have the task return to your computer should the need arise. This would simulate the absurdity above to a large degree. *Mobile agents* allow just this: a mobile agent is a software agent that is not bound to a specific computer, but can be moved to another machine and continue its task there.

We present mobile agents as a new and powerful paradigm for distributed computing, and describe what the benefits are of using mobile agents as the

[1] Unless the task you were thinking of was something like 'take up space on my desk', but let's not go there.

foundation for a distributed computing architecture. Architecture in the sense that mobile agents are taken as the fundamental building blocks of the application, possibly together with normal (non-mobile) agents and normal objects, both when modelling and implementing the application.

Applications that could benefit most of this paradigm are applications that need to:

- undertake activity without a user actively interacting with the application
- arbitrarily connect and disconnect machines and users from the network
- manage bandwidth and availability differences
- have massively parallel computing

1.3 Goals

In this thesis, we investigate what software agents are, what makes them so special, what they can be used for, and how to think of computer applications in terms of agents. We do not focus on any specific application of software agents, but rather on agents as a concept, as a computing model.

goals

We set ourselves the following goals:

> We want to present *mobile agents* as a *distributed application architecture*, as a paradigm for building software systems running on interconnected computers. We want to present *everything agent specific necessary to build such an application*, so we discuss what agents are, what mobile agents are, how a (distributed) software system can be designed and modelled using agents, and how an agent based system can be implemented.

In the next section, we detail how we cover these aspects in the various chapters.

1.4 Chapter outline

This thesis only assumes a basic understanding of how computers work and what computer programs are. We try to explain many of the more advanced and/or agent specific topics thoroughly. However, do bear in mind that some material might be quite dense if you are new to areas such as object orientation and programming. Those who live and breathe agents in everyday life can skim through many areas quickly, but we do believe those might still find some interesting new ideas.

We don’t have the illusion that everyone who reads this thesis will do so from front to back. This outline should give you some idea of what is of interest to you and what is not.

- **chapter 1**, this chapter, introduced what is discussed in depth in this thesis.
- **chapter 2** defines what agents are, shows agents as a computing model, discusses some common types of agents, and shows what could go on inside an agent.
- **Chapter 3** investigates intelligent agents in depth.
- **chapter 4** shows how agents can be used in an OO design, and how to look for agents when designing an application.
- **chapter 5** introduces the mobile agent paradigm. It discusses what is so special about mobile agents, what they can be used for, and how mobile agents compares to other types of distributed computing. It concludes with some ideas on how mobile agents can be used for Internet applications and how mobile agents can be used to create a powerful application server platform.
- **chapter 6** discusses how agents and mobile agents can be implemented, using what technologies. It shows different models of agents, different models for agent autonomy, a simple design pattern for mobile agents. It includes sample implementations in Java.
- **chapter 7** describes the design and implementation of a two-part prototype application: The Octarine mobile user agents framework and WebWatcher. The framework is capable of supporting user agents that are able to move between the user’s machine and a central server. WebWatcher is a front-end application for this framework.
- **chapter 8** gathers the most important conclusions drawn from this thesis, and discusses some future directions.

If we may present some guidance:

- For more business oriented people interested in agents and mobile agents, chapter 2, chapter 5, the beginning of chapter 7, and chapter 8 would be of most interest.
- High level technical people would additionally want to read chapter 3, chapter 4, and the remainder of chapter 7.
- Finally, developers, especially aspiring developers, should just read the whole thing.

2

CHAPTER 2 Agents

You probably have a pretty good idea of what an agent is in real life. Most of us interact with agents every now and then. You arrange travel via your travel agent, insurance hassle is taken care of by your insurance agent, and her Majesty's little errands are run by a secret agent. If you have already made your way into the land of fame and fortune, you might have an agent taking care of your busy schedule and answering questions.

The topic of this thesis concerns agents in the realm of computer science, in their most general form usually referred to as 'software agents'. As with many relatively new terrains of research, there exist just about as many different definitions of (software) agents as there are papers about the subject. These definitions range from a very strict definition of the behaviour and algorithm of an intelligent program, to a mobile process, to the behavioural view saying that an agent is anything that is convenient to call an agent (for some examples, see [1]). Existing applications range from a flexible architecture for a digital library [11] to animated characters on a screen that have a vocal conversation similar to humans, including change of intonation and hand gestures [12].

Given all this confusion and controversy, we think it is imperative we try to give a good, solid definition of software agents. Doing so will not be easy however, because the concept 'software agent' is at its core very abstract, often used in complex settings, and very flexible in morphing itself to fit into the context. Sometimes it seems an agent is more like a feeling than something you can put your finger on.

The main reason anyone would want to refer to a particular piece of software as an agent has to be because it resembles a real-life agent in some way. Therefore, before we look more deeply at software agents, let's see what the dictionary says about agents in general.

2.1 Agent according to the dictionary

agent

The Oxford English Dictionary of Common English defines 'agent' as follows:

agent […] n. **1 a** person who acts for another in business etc. **b** spy. **2** person or thing that exerts power or produces an effect.

Webster's Revised Unabridged Dictionary more or less agrees:

agent […], n. **1.** One who exerts power, or has the power to act; an actor.

[…]

2. One who acts for, or in the place of, another, by authority from him; one intrusted with the business of another; a substitute; a deputy; a factor.

3. An active power or cause; that which has the power to produce an effect; as, a physical, chemical, or medicinal agent; as, heat is a powerful agent.

to act

The word's origin stems from the Latin verb 'agere', meaning 'to act'. Also, both dictionaries[2] define the adjective agent as 'exerting power, as opposed to patient'.

We thus have two somewhat different meanings of agent: someone representing or acting under the authority of someone else, and someone or something exerting power, producing effects. The 'spy' meaning of agent also falls under the first. After all, James Bond is acting under the authority of her Majesty the Queen herself.

autonomy

The second meaning says an agent has the power to act. This implies an agent has some degree of *autonomy,* it can do things on its own. In fact, we can say the first definition of agent is enabled by the second definition: An agent has the power to act, and as such is able to (autonomously) represent someone else or act under the authority of somebody.

This autonomy is the main reason businesses let themselves be represented by agents. The agent can dedicate his or her attention to a (potential) client, while the people running the business can do just that: run the business. The agent might even be at a different location.

And the knife cuts both sides. An agent ought to have extensive knowledge of the offerings of a particular business, and can assist the client in choosing the service most suited for him. This is especially important if the service is complicated and/or tailored to the client's needs. The agent thus also becomes the representative of the client to the business, especially if the agent is 'independent', representing several businesses with similar offerings, such as insurance.

Often the same agent has repeated interaction with the same client. As such, the agent can maintain a history of a client, which allows it to easier match new requests to the client's needs and preferences.

[2] For Oxford's, this is in the full, historic dictionary.

2.2 Why agents are important

attention

Agents become increasingly important, and consumers gain a lot by having agent-type of services available. Given the current wealth of information, *attention*, not information, becomes the most important asset [13]. Why? Well, there are only 24 hours in a day, of which we sleep at least a few. During those remaining hours left we are overwhelmed by an avalanche of information (emails, memos, advertisements, billboards, the news etc.), and are able to access even more (the newspaper, the corporate data store, archives, the Internet etc.). More and more your daily task becomes very carefully distributing your attention amongst many different information sources. Advertisers for instance know this and try ever crazier things to get your attention. Perhaps the wildest one we know is a cola brand that had people rake out its trademark logo in the beach every morning so the beach goers would see it.

delegation

By *delegating* certain things to agents, you can reduce the amount of attention required to find just that piece of information you need. Take something like picking a restaurant to eat with a good friend. If you pick a wrong restaurant, that's a double waste: you have bad food, and you waste time that should have been quality time spend with your friend. Now suppose you make your choice based on the recommendations of a food reporter you like. This dramatically increases the odds you pick a good place, because: a) the food reporter has already visited a lot of restaurants so he or she knows what he or she's talking about, and b) because you know you tend to agree with the food reporter. This all in substantially less time than it would take to scan the entire list of local restaurants in the Yellow Pages. In this example, the food reporter is effectively your agent, representing you (and many others) when he or she visits and rates restaurants on your behalf.

2.3 Software agents

The way we want to define software agents now flows naturally from our discussion in 2.1. We define a software agent as follows:

software agent

A **software agent** is an autonomous software component that perceives, reasons and acts.

perceive, reason, act

We already saw 'autonomous' and 'act' in 2.1, and, like a normal agent, a software agent is able to act, and is able to do so autonomously. In order to act sensibly, the agent should *perceive*, *reason* about what it just perceived, and *act* based on those perceptions now or later. Finally, 'software component' means the software agent can be stand alone, or be part of a larger application, which could very well consist of several software agents. Of course, this software component needs to run on some computer for the agent to be able to do anything.

From now on, we will often omit the 'software' from 'software agent' for brevity. It should be apparent from the context however, when we refer to a real-life agent or its software counterpart.

2.4 The importance of autonomy

Arguably, just about all software out there perceives, reasons and acts to some degree, but the autonomy aspect is what makes the agent special. It enables the agents to make decisions on its own, in particular without user intervention. While most software is centred on serving a user or other systems, agents are more self-centred, more, as said so many times now, autonomous. The autonomy enables us to really *delegate* a task to the software agent, just like we delegate a task to a real life agent.

asynchrony

Another important aspect of autonomy is that it enables the agent to exhibit asynchronous behaviour. Asynchrony is a very desirable feature because it allows for greater concurrency and flexibility. It is a bit like the difference between having a conversation over the phone and exchanging email. A phone call requires both parties to pay attention at the same time and respond immediately, while with email each party can respond at a time he or she sees fit, if at all.

Still, autonomy is not some magical property we need to give the agent. Any computer program is potentially capable of showing autonomous behaviour, they just rarely do so. Some examples that do, are screen savers and insuspicious background processes like mail daemons. These can be considered very simple agents.

2.5 Roles for agents

It is often confusing what a software agent is exactly. Many, at first glance unrelated applications are described as being agent based or using agents in some way. In addition, we believe there are pieces of software out there that could be thought of as agents even though they are not labelled as such. We read many articles before we grasped the core of the concept and came up with the broad definition as described above. However, this definition doesn't give many clues to what agents are useful for. Below we list a number of roles agents typically play, along with some other possible names for an agent in that role, some characteristics, and examples of applications that use agents in such a way. This list is by no means meant to be exhaustive and there could very well be agents out there that play a different role than described here. In addition, one of the powers of the agent concept is that it is very natural for an agent to have more than one role. Nevertheless, together these roles give a nice indication of the possible uses of agents.

2.5.1 Agent as an autonomous entity

Other name for this role: Actor

This is somewhat the mother of all other roles, as apparent from the definition above, but it is listed separately because the autonomy is sometimes the most important property of the agent. This role often occurs in simulation or animation, where the agent represents a character in the system, pursuing it own goals, often interacting with other agents.

Examples are for instance two fully animated characters on a screen, engaged in a social interaction [12], and Lyotard, a simulated cat [7].

2.5.2 Agent monitoring input

Other names for this role: Monitor, Alerter

This is a very simple agent that watches some source of data, like a database table, or some physical device, and the agent's actions consist of sending some messages to someone or something. The agent could be sending an email, calling a pager, or placing a message in a message queue. Someone or something then takes action on the message sometime or later.

Examples include: Agents that monitor network status and alert operators of network problems, or web sites that send out update notifications such as e-Bay [22].

2.5.3 Agent controlling something

Other names for this role: Controller, Manager

This is a beefed up version of the Monitor role, where the agent in addition takes significant actions based on the input, instead of just sending notifications. This is especially useful if the actions to take need to be taken very quickly and/or the decisions require expert knowledge (This of course requires that this expert knowledge is obtained and stored with the agent).

Examples include several agents together controlling a particle accelerator [15], or agents deciding which machine job to process next [16].

2.5.4 Agent providing a service

Other names for this role: Service, Provider

Here we see that the knowledge of an agent is very important. The agent knows how to do something, and provides it as a service. This model is a good basis for a flexible application architecture where various agents hide the complexities of different systems or functionality and present a common interface. It differs from a more traditional client-server model in that the agents are not purely passive like normal server components, but actively do something as part of their task (i.e. monitoring and indexing a database). In addition, the agents might accept tasks that are executed asynchronously.

Examples include the open, agent based architecture of the University of Michigan Digital Library [11], or task bots that can do secretary-like tasks [17].

2.5.5 Agent helping a user

Other names for this role: (Personal) assistant, Guide, Teacher.

This is one of the better known and most visual roles for an agent. The agent provides assistance to the user in the form of explanation and advice, often as an animated character on screen.

The notorious paperclip (Office Assistant) in the Microsoft Office applications is a well known example, another, less famous one, is Herman the Bug in Design-a-Plant, teaching middle school children about plant anatomy and physiology [18].

2.5.6 Agent representing someone or something

Other names for this role: Representative, Proxy.

Very handy: having an agent represent you, so you don’t have to do something yourself. It could also be the other way around: a piece of software on the user’s machine can be some business’ agent, and can tell the user when something interesting has occurred.

As for the first type: some more advanced email packages, such as Microsoft Exchange, have an ‘out of office assistant’ that reply any incoming email message with automatically while you are gone. These messages can differ depending on the type of message. For instance, a message labelled ‘important’ might get a reply with you private mobile number, whereas a normal one just states you’re away, and when you will be back.

An example of the latter is the ‘full circle’ agent in the newer Netscape browsers that pops up when the browser accidentally crashes. The agent asks the user to (optionally) fill in a problem report to send back to Netscape.

2.6 Agents in depth

We’ve seen what an agent is and what roles it can play, now we want to examine more closely how an agent actually works, how it can perceive, reason and act.

2.6.1 An agent and its environment

environment

An agent always perceives and acts on its *environment*. This environment might be a real physical environment, so the agent perceives light or sound, and its actions have a physical effect like the movement of an object or the display of a message on a screen. A robot can be thought of an agent in a completely physical environment.

The environment could also be virtual, so only existing in a computer or computers. Monsters in a 3D-action game can be seen as agents in a virtual environment. A subtle point arises here: This virtual environment is projected on a computer screen, so an agent's actions on a virtual environment might still have a physical effect.

Finally, the environment might be a mix of both. For instance, an agent that teaches a user something in a multimedia application, communicates with that user (physical), but also observes the user's actions on the taught subject (virtual).

To avoid complexities we will assume that as far as the agent is concerned, its environment is what it can perceive from and/or act on. If we initially treat the agent as a black box[3], we get this very simple model of an agent in some environment.

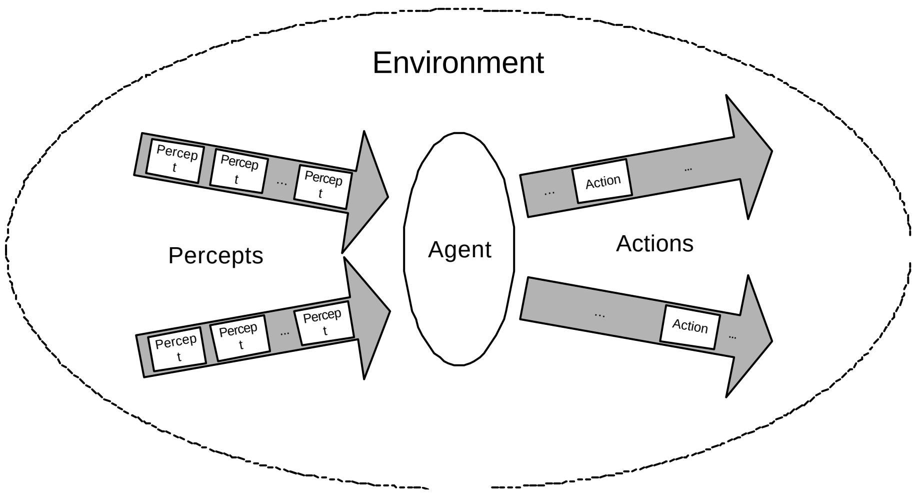


Figure 1: A first model of an agent.

percepts, actions

We see two streams of *percepts*, things that are perceived, going into the agent, and two streams of *actions* leaving the agent. Two of each, to indicate the possibility of many independent streams of different types of percepts and actions. The percepts and actions occur in time. The agent processes percepts arriving in time, reasons about them and acts every now and then.

neuron

Especially those who are familiar with biology or neural networks might find that this model resembles that of a *neuron* (nerve cell). A neuron also accepts input (electrical pulses) and fires (acts) depending on the values of the input. A software neuron as used in neural networks (see [14]) is a simple software model of a real nerve cell. The difference between a software neuron and a software agent is a matter of scale and complexity. In a simulated neuron, the input and output are of the same type, and the transformation is relatively simple. For an agent however, the inputs, actions and the reasoning process can be of arbitrary complexity.

[3] Depicted by a white oval. (pun intended)

2.6.2 State

state

Without change, the world would be pretty boring. Most things around us change as time passes. The weather is rainy and cold at one point in time, sunny and hot at another. A light switch can be on, or it can be off. Each of the conditions described here is a *state*. So the light switch is in the state on, or in the state off. Under normal circumstances, something is in only one state at any give point in time.

stateful, stateless

Something that can be in more than one state is called *stateful*, something that cannot is called *stateless*.

With the concept of state, we can now describe a little more precise how an agent works. When an agent perceives, it perceives the state of its environment. Some of the states of that environment will cause the agent to take action, and the taking of that action in turn changes the state of the environment. In our discussions, we will assume that the agent gets feedback on its actions, that it is able to perceive the result of its actions, whether successful or not. This is not always necessary in practice, but it makes describing how an agent works somewhat simpler.

2.6.3 Reactive and proactive

reactive agent

So far, only the environment had state. The agent itself was stateless. Such an agent is a *reactive agent*: It immediately responds with an action once its environment has a certain state.

What happens if the agent has state as well? The agent can, instead of affecting its environment, change its own internal state, so the agent's perceptions do not have to have an immediate effect. But when something else happens later on (which might just be the passing of a certain amount of time), the agent can take an action based on the earlier perception and the current state of the environment.

proactive agent

This kind of behaviour is called proactive, and an agent that exhibits this kind of behaviour is a *proactive agent*. A simple, real life example: A stop-loss order. You tell your stockbroker to sell certain stocks if they sink below a certain level. Your broker does not sell the stocks immediately, but he or she would once they sink below the threshold.

A proactive agent is capable of much more complex behaviour that a stateless agent. It can combine events that happen at different points in time, which means it could for instance accept new tasks to do, and even 'learn'.

2.6.4 Inside an agent

Now it is time look at the inside of an agent. What goes on inside the agent that makes the agent do its tasks? Software agents derive their name from their similarity to human agents, so it should not come as a surprise that a good way to model the inside of an agent is similar to a model of a living being. Specifically, our model consists of a *heart*, *sensors*, *effectors*, a *brain*, and *memory*. In a picture, this looks like this:

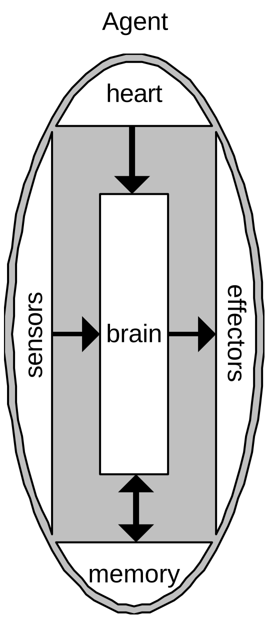


Figure 2: Inside an agent.

This picture mimics the same oval for an agent as shown before, but now with the inside filled in. The percept and action streams are omitted for clarity, but you can think of them in the same place as they were in Figure 1. The black arrows indicate which component influences each other. What happens is described below:

Heart
The heart is the source of the agent's autonomy. It periodically activates the agent's brain, and that activation might cause the agent to do something. How the heart is implemented determines how autonomous the agent is with respect to its environment. For the technical details of this, see 6.3.

Sensors
The sensors provide the agent with information about its environment, the percepts mentioned earlier. For a software agent, this might be data from a file, the current time, input from a hardware device (thus a real 'sensor'), data coming from a network, information received from another software component, etc.

Effectors
Effector is a general name for 'something to influence the environment'. For a living being these thus are arms, legs, paws, beak, vocal cords, etc. For an agent, its effectors might enable it to write to a file, manipulate a hardware device, send data across a network, send information to another software component, etc. The effectors thus determine what different actions an agent is able to take.

Brain
The brain allows the agent to reason, to decide if and what actions to take. It combines information from the sensors and the memory to activate some effectors and/or update the memory.

Memory
The memory provides the brain with historic information and basic knowledge. We have separated memory from the brain to distinguish between a component that does the thinking, and a component that stores information. The memory allows the brain to store current information for future reference, it provides the agent with *state*.

agent compared to a computer

We think this model pretty accurately shows how a living being interacts with its environment, and also how a software agent should interact with its environment. You might have the eerie feeling this looks like nothing new. Indeed, if you replace ‘heart’ with ‘clock’, ‘brain’ with ‘CPU’, ‘sensors’, with ‘input devices’ (keyboard, mouse, etc.) and ‘effectors’ with ‘output devices’ (monitor, printer, etc.), you get a simple model of a computer running a program. So what’s different? Well, most importantly, the concepts of sensors and effectors are meant to be much more general. They could be sensing in a partly or completely virtual environment. Also, we consider many different agents on one computer, even per application.

2.6.5 Agent cycle

agent cycle, beat

We’ve been saying that an agent continuously perceives reasons and acts. While it is definitely possible to do these things in parallel, there is a natural order in these three elements, which is also apparent from Figure 2. It starts with perception, followed by reasoning (including a possible update of the agent’s memory) and ends with taking actions. We will call this the *agent cycle*, and a single iteration of this cycle a *beat*, to reflect that the execution of the cycle executes in a single ‘beat’ of the agent’s ‘heart’.

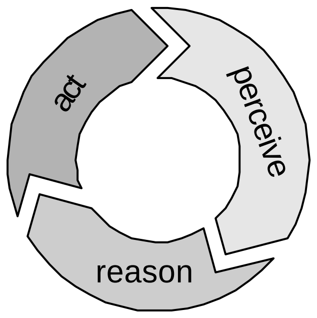


Figure 3: Agent cycle: perceive, reason and act.

2.6.6 Multi-agent systems

multi-agent system

Although a single agent by itself can already show very complex behaviour, it gets even more interesting if we consider multiple agents interacting with the same environment. Such a system is called a *multi-agent system*.

Multi-agent systems are big part of agent research by itself. (See for instance [15]). Multi-agent systems are for instance a way to approach the study of complex, dynamic systems like trade economies or schools of fish. The big advantage is you can take a bottom up approach, starting with the simple components of the system (the trader or the fish) and describe their behaviour. This is often much easier to capture than the behaviour of the system as a whole.

2.6.7 The lifetime of an agent

hibernating agent

Different agents generally have different lifetimes. At the one extreme, we can have agents that live ‘forever’, that remain running as long as a computer system is up, agents that provide a core system service. At the other end we can have agents that execute a certain short task, agents that only ‘live’ for a few seconds. It might also be the case that the certain agent ‘hibernates’ for some time, with its state stored on disk, but without the agent being active in the memory of the computer.

For instance, the article *An Experiment in the Design of Software Agents* [17] makes the very useful distinction between user agents and task agents. (There referred to as user bots and task bots). A user agent not surprisingly interacts with a single user, probably using some sort of GUI or Web interface. A task agent on the other hand, continuously executes a fixed task, such as monitoring a data stream or, as in their case, scheduling meetings. The user agents come up and come down when the user logs on logs of, whereas the task bots run continuously.

2.7 Agents: a different model of computing

The good thing about our definition of agents that it catches what we believe to be the core of agency. The bad thing about it is that it might also cover pieces that are not normally considered agent.

web server vs. agent

Consider for instance a web server. A web server continuously listens to a network port, and when a request comes in it finds the requested page and sends it back to the requester. This can be interpreted as perceive, reason, and act, though in a very limited way.

Does this mean a web server an agent? No, not really. The difference lies in the fact that all interaction with the web server is synchronously. You send a request, and wait for the web server to satisfy it. There is a one-to-one mapping between input and action (output), and the output always goes to the requester. The same is not true for an agent. The agent does not necessarily send a response back to the source of the message, and whether it takes action at all often depends on previous input (possibly from another source), and if it sends back results it often does so asynchronously.

Software agents as we defined them in this chapter present a different computing model. The traditional model of computing is that of input-transformation-output, as with the mobster and his goon: 'You're not paid to think. You're a muscle. Relax until I say 'flex''.

more interactive computing

Agents are a way to think about breaking this model, to go towards a more interactive way of computing, where human and computer are in a dialog to accomplish a task, and the human can delegate some mundane and / or highly specialised tasks to the computer. Additionally, it could be the computer starting the dialog, instead of just the human. And this way of thinking is not limited to human-computer interaction. It is also a very useful way of thinking of a large (distributed) application, that, instead of just clients and servers, also has several agents that are assigned particular tasks.

Partitioning an application in several agents makes it easy to define a task for each agent, and probably only one of them would be responsible for communicating directly with the user

Most traditional software modelling methods consider the system to be modelled to be largely reactive, to respond to outside stimuli only. This makes it hard to model more interactive systems and systems that exhibit asynchronous behaviour. With computer systems becoming ever larger, this becomes increasingly apparent. Modelling using agents is a possible way of tackling this problem, simply by making different agents the source of the asynchronous behaviour. We will introduce this approach in chapter 4, where we integrate agents into the basic modelling constructs of UML, an object oriented notation language and methodology.

2.8 Types of agents

Now we know a little more of how an agent works, we want to discuss different types of agents that are commonly recognised in the agent literature. This is not categorisation, but each type highlights a particular additional property an agent might have. For these and other types of agents, see also [1].

2.8.1 Intelligent agents

intelligent agents

A large part of agent research concerns *intelligent agents*. Roughly speaking, an agent is intelligent when it can act 'smart' in many different situations. Intelligent agents generally deploy some form of Artificial Intelligence (AI) techniques for their reasoning, such as neural networks, genetic algorithms, decisions trees or logic programming. In fact, the book *Artificial Intelligence; A modern approach* [2] states that 'The purpose of AI is to constructed intelligent agents.'

Intelligent agents are not the direct topic of this thesis. Nevertheless, we devote the next chapter entirely to intelligent agents, because we think it helps illustrate the importance of agents, and really exemplifies what makes agents so special.

2.8.2 Mobile agents

mobile agent

Mobile agents are another big part of agent research. (See for instance [27] or [29].) A *mobile agent* is an agent for which it is possible to move or copy a running instance to another computer in a network. This ability to move is a possible, rather fundamental action the agent is able to take while running, one which can have a rather big influence on what the agent is able to do next. Actually, the mobility of the agent is so important it often overshadows the ‘agentness’ of the mobile agent. A mobile agent is a mobile, autonomous component, and the sensing, reasoning and acting are less important. A mobile agent is above all a courier, an autonomous component running around and doing different things in different places.

This mobility creates some exciting possibilities, such as being able to configure the agent on a user’s machine, and then move it to a central server. This would allow the user to disconnect from the network, while the agent would continue to do its task. Once the agent decides it should alert the user, it could move back to the user’s machine (perhaps waiting until the user is connected) where it could interact with the user and probably present some data it has found.

load balancing, fail-over

Another idea is to have the agent manage server load and fail-over by moving to a less-loaded server and distributing copies of itself across the network. This gives a hook for a highly decentralised and thus very scalable approach to load balancing and fail-over.

mobile agents on the Internet

In its most general and unrestricted form, mobile agents could travel all over the Internet, interacting with servers and other (mobile) agents. This by itself interesting idea has some serious consequences in terms of security and responsibilities. Because of these issues, we look at the implications of restricting the mobility of the agent, in particular between machines in a trusted network and between the trusted network and a user’s machine.

We explore these topics in much greater detail when we discuss the mobile agents paradigm in chapter 5.

2.8.3 Social agents

Many agent systems are multi agent systems (see 2.6.6), they consist of several agents. Although each agent could mind its own business, often the agents communicate with each other. This might be indirectly, by one agent observing changes in the environment made by another agent, or directly, if there is explicit inter-agent communication (see 2.9). If those agents are able to co-ordinate and assign tasks to one another, you create a society of agents. Co-operative problem solving is one of the most important capabilities of agent-based systems [1].

Although communicating agents are often called social agents, the agents might not be social in the 'friendly' sense of the word. Agents are often constructed to be self-interested, to pursue their own, selfish goals. This might mean the agents only co-operate if it is in their own benefit, and even that agents lie and deceive other agents to best pursue their own goals.

It is often desirable for the system as a whole to have different agents speak the truth to one another. This poses interesting challenges for the construction of the agents and for engagement protocols that minimise the gains one can get by lying. For some examples, see [31].

2.9 Inter-agent communication

Systems consisting of several agents often require that those agents are able to communicate with each other. If those agents live in the same application on the same machine, this can probably be accomplished in pretty much the same way as other data is handled in that application. However, if those agents exist in different applications on different machines, you need a standard and well-defined way of communicating. There has been quite a lot of research in this area, especially because the agents are often intelligent (see 2.8.1) and they need to exchange knowledge with each other.

KQML

The Knowledge Query Manipulation Language (KQML) [20] is the current standard method for exchanging knowledge among agents. KQML defines a set of standard primitives to inform other agents of some perceptions of the agent, to ask other agents to take some action, and some more. KQML also defines the protocol of how the communication takes place, but it doesn't define how the data is represented. This is delegated to a knowledge representation language, such as KIF.

KIF

The most used language for knowledge representation among agents is KIF, Knowledge Interchange Format [20]. KIF is an extended form of first order logic, equipped with sufficient expressive power to express knowledge commonly found in knowledge bases, and suitable for automatic translation to and from internal knowledge representations.

ontology

Event though two application (agents) have a way to exchange knowledge, that does not mean they agree about the semantics of the knowledge, i.e. what that knowledge means. A good way to solve this is to use an ontology [20][21]. An *ontology* is a standard database for a given knowledge area. Using the common ontology the two agents can make sure the receiving agent indeed receives the knowledge the sender intends it to receive, without the risk of misinterpretation. Each agent might know how to translate its knowledge into one or more ontologies. To send a message, the agent first finds out if there is an ontology both parties can translate to. Then translates the knowledge it wants to send from its own internal representation to that of the chosen ontology, and sends the message. The receiver accepts the message, translates the knowledge in it into its own representation, and adds it to his knowledge base.

KIF has a core set of primitives with known semantics, and it has a mechanism for defining new language elements based on already defined elements. So in theory, two agents using KIF could do without an ontology. However, it is tedious to do so at each exchange, and it might even be prohibitively difficult for some element, so even applications using KIF could benefit from using an ontology.

Ontologies thus allow two completely different agents on different platforms with different implementations to communicate in a well-defined way. This approach has been very successful in integrating previously independent systems [21]. Instead of painstakingly defining a common database for data representation and adapting each application to use the new model, each application is equipped with an agent that monitors the data of that application, and knows when to send updates to the other applications. Beforehand, one investigates what data in each system is of interest to other systems. That (relatively small) set of data is defined in a common ontology, and the agents communicate with each other using that ontology.

2.10 Sample agent applications

We want to conclude this chapter by describing three existing agent applications as found in the literature. Each is a good example of how agents are used to solve a particular problem. We start with PACT, were agents are used as a systems integration method, then on to an intelligent email assistant that has friends to help him, and finally an initiative to put mobile agents in your mailbox.

PACT

designing a robot

The Palo Alto Collaborative Testbed (PACT) is by now an almost classic example of how agents can make a difference. PACT is a research initiative to ingrate several engineering systems [32]. The design of a complex system like a robot is a multi-disciplinary problem. One of the initial experiments with PACT dealt with the design of a robotic manipulator. It involved the physical design of the manipulator, and the design of digital circuitry, analogue circuitry and software. Each of these four tasks is a specialist’s discipline, and the design of it is done by a separate engineer or engineers. Each of those engineers uses specialised design software to design his or her pieces of the manipulator.

Of course, choices made by one engineer affect another. If the physical design of the device involved four motors, the analogue circuitry should be able to power those four motors. Without any integration of the different design tools, these things have to be communicated among the different disciplines by for instance joint meetings or email. It goes without saying that this is a slow and error-prone process. In addition, to really prove the design works a prototype has to be built, because only then we could see if the different pieces of the design really work together.

It should be clear that integration of the different software tools could provide substantial benefits. Changes made by one designer would be immediately apparent by another designer, who can then instantly redesign his or her piece to accommodate the change. It could even be possible to run a simulation of the design as a whole, which no doubt is much less costly than the creation of a physical prototype.

So how does one go about integrating these different systems? The ‘traditional’ way of doing this would be to create a unified, shared model database and adapt each of the software tools to use that database instead of their own. This is certainly possible, but very difficult. Each of the tools has their own, very specialised needs of the model. For instance, for the physical model, the model of a motor concerns mainly what forces it is able to produce, the precision of the movements, and the size of the motor. The analogue circuitry designers on the other hand, are more interested in what voltage the motor needs, and if there are non-linearities in the power-input to force-output function of the motor. And there are reasons that are more technical as well. Each of the design systems runs on different, often specialised hardware, and the local model database is optimised to allow for fast modelling. Connecting the modelling system to a unified, less optimal database on a separate machine could bring performance to a grinding halt.

For these and other reasons, the researches chose a different approach for PACT. Each of the design tools was equipped with an agent that monitored that tool’s database, and reported any changes to a facilitator agent. The facilitator agent would know if there were other tool agents that were interested in the change, and if yes, report the change to the appropriate facilitator agents. In turn, such an agent would then tell the local tool’s agent, which would update its internal model database. This creates a clear separation in each of the agent’s tasks. The tool agents were responsible for monitoring and updating their own database, and the facilitator agents enabled those agents to communicate. The tool’s agent could speak its own ‘dialect’ to the facilitator agent, which would translate the change (if any other agent was interested) in a common language used between the facilitator agents. The receiving facilitator agent would translate the message into the dialect of the local tool’s agent.

The challenge of course was to design the common language used by the facilitator agents. In this case, the facilitator agents use KQML as a communication language. KIF was used as the content language, with a vocabulary based on common ontology for the shared application domain. (See 2.9)
You might think that defining the content language is just about as difficult as designing a unified database. This is not true, for two reasons. First, the only things that need to be agreed upon are the items in the designs that affect more than one model. For instance, any chips used in the digital circuitry are of concern only to that area. Second, even if there is an overlap, the only things that need to match are a way to identify a particular part and some very basic parameters. There is no need for a shared model that matches all the requirements of all the tools. For instance, if the physical system designer decides to pick another motor, the only thing the other analogue system designer has to know is the part number of that

motor, essentially the same as if the designers would communicate by email or in a meeting.

Collaborative interface agents

One of the more useful aspects agents from a user's perspective, is have agents take care of mundane tasks. The article *Collaborative Interface Agents* [33] describes an agent that integrates with the Eudora mail client on the Apple Macintosh. The agent is able to control the mail application in much the same way as the user, so it can move mail to different folders, delete messages, print, etc. The Eudora interface is enhanced to allow the agent to do things: suggest an action on some message, or execute a certain operation immediately, without asking. Which of the two is chosen depends how confident the agent is the user will really take the action. Of course, execution without asking first takes more confidence. These confidence levels are user definable.

There are roughly two ways this type of agent can be implemented: having the user define a set of rules for the agent, or have the agent learn form the user's actions, by sort of 'looking over the shoulder' of the user. The first approach has the advantage that it can be directly controlled by the user, and the user has a clear picture of what the agent does. The disadvantage is generally takes a skilled user to define these rules, and the user first has to figure out what he or she does with certain messages. The learning approach means the agent acts like an adaptive, intelligent agent, gradually adapting to its environment, and gaining more confidence in its predictions. This has the advantage that the user doesn't have to do any rule definition, and that the user can gradually get used to the agent, and learn to trust the agent's predictions. The big disadvantage is that the agent has to learn from scratch, which means it might take a long time before the agent can make accurate decisions.

The agents described in the article are built using the learning approach, but with a nice feature to flatten the initial learning curve. It can communicate with 'older' more experienced agents in the system (the agents were deployed on a campus system), and ask them what their response would be in a certain situation. This happens for two reasons: out of desperation, and for exploration.

Desperation is pretty straightforward, the agent does not have a clue what to do in a certain situation, so it asks what a peer would do. This allows even a very new agent to get a shot at making accurate predictions, immediately providing value to the user.

Exploration happens when the agent is idle and it asks a peer what it would do in some situation. In this case, the agent *knows* what the user's action was for the message, so it can, based on the response of the other agent, gradually decide how much trust it can put in predictions of that agent. Experimentation has shown that a collaborative version of the agent was much quicker able to make accurate predictions than one that just had to learn on its own.

Active Email
Normally, email messages are largely passive, they do not interact with the user or the mail system in any big way. Email could very well be even more useful if it allowed for some type of interactive content. Users of Microsoft Exchange systems have limited access to more active email messages. For instance, when you want to schedule a meeting, you create a meeting entry on your calendar, and specify what other people you want to attend. Each of those attendees then gets an email message with ‘accept’, ‘decline’ and ‘tentative’ buttons. Pressing one of those buttons then allows the user the choice to send a response straight away, or to edit the response first.

Safe-TCL

The article *Email with a Mind of its Own: The Safe-Tcl Language for Enabled Email* [10] proposes a generalisation of this concept, allowing the inclusion of pieces of code in an internet email message that get executed at send time, delivery time and / or opening time. Effectively, such an email message can be seen as an application of mobile agents. The message is an agent, that is able to exhibit autonomous behaviour at several different times during its travel among different sites. This construct allows a nice way to send things like questionnaires or even presentations. It is another, rather elegant form of so called push-content.

Because the code executes in an email context, it is much easier to allow the user control over responses that get send to the original sender, and for instance a questionnaire based on this idea would be much easier to set up than the equivalent web implementation.

The active mail is created by inserting the active content as a MIME attachment in the message. The active content itself is writing in a language called safe-TCL, a trimmed down, safer version of TCL[4], short for Tool Command Language, a scripting language most seen on UNIX platforms. This ‘dialect’ of TCL is created by removing possibly harmful primitives (most notable those for manipulating files) and replacing those with safe ones with limited functionality. Additionally, it defines some additional primitives to allow for easy, generalised interaction with the user, and sending of new email messages.

2.11 Conclusion

Software agents are a tough concept to grasp and define. We have chosen to define it a very general manner, as an *autonomous* software component that *perceives*, *reasons* and *acts*. While any piece of software can be thought of to perceive reason and act, the autonomy is what makes such a piece of software an agent. Where a ‘normal’ software program is largely used in a request-response setting, an agent is more self-centred, and decides *if* and *when* to respond, rather than just what the response will be.

Software agents always reside in some *environment*. As far as the agent is concerned, this environment resides in a computer or computers, although

[4] TCL is sometimes pronounced as ‘tickle’ in the same way SQL is pronounced ‘sequel’ for some silly reason.

the agent’s perceptions in this environment might have a source in the physical world or the agents actions have a physical effect. When an agent perceives, it perceives the *state* of its environment, which changes over time. If an agent decides such a change is significant, it will take an appropriate action. If the agent can only take this change immediately, the agent is a purely *reactive agent*. If the agent can have different states as well, it can ‘remember’ things. A set of perceptions at different times can lead to a certain action, or the agent can decide to take multiple actions after another. Such an agent is called a *proactive agent*, because the actions of the agent are not just the result of the current state of the environment.

An agent is often not alone in its environment, but shares it with other agents. Such a system is called a multi-agent system. Those agents might be on the same computer, but they don’t have to. If they are not, such a system is thus a form of distributed computing, in particular suitable in situations where there is not clear client-server relationship between the different computers. Because a multi-agent system consists of many autonomous components, it is capable of showing extremely complex dynamic behaviour.

Inside an agent, we recognise five different components:

- The heart, which gives the agent its autonomy,
- sensors, allowing the agent to perceive,
- effectors, allowing the agent to act,
- brain, allowing the agent to reason,
- memory, allowing the agent to remember and store things, thus giving the agent state.

These components influence each other in a certain way, which gives rise to the *agent cycle*:

1. Perceive,
2. Reason,
3. Act.

We call a single execution of this cycle a *beat*.

We recognise three important types of agents: intelligent agents, mobile agents and social agents.

Intelligent agents
Intelligent agents are agents that are able to respond in a ‘smart’ way in many different situations. The brain of these agents is generally based on some form of artificial intelligence.

Mobile agents
A mobile agent is an agent for which it is possible to move the agent, in its current state, to another machine, where it resumes execution. This is a

fairly new and novel model of computing, which opens up some exciting possibilities. The mobility makes the agent into a distributed computing concept, which often overshadows the other aspects of the agent.

Social agents
Agents in a multi-agent system that explicitly communicate with each other are called social agents. Social agents can ask each other agents questions or ask it for help. Agents are often self-interested to some degree, which means that they might not tell the truth to one another.

It is often useful to have the agents communicate using a well-defined language, such as KQML/KIF. The use of an ontology, a sort of ‘common’ dictionary, can help avoiding ambiguities in the inter-agent communication.

CHAPTER 3 Intelligent agents

A very well known type of agent is the intelligent agent. Interestingly enough, intelligence in relation to software is rarely defined in a solid manner. It is sometimes related to human intelligence [1], but we will avoid this since a computer program that is just as intelligent as, say, an ant, can still be very useful, and, on the other hand, humans have been known to do pretty dumb things occasionally.

intelligence

A definition that we prefer is:

> **Intelligence** is the ability to adapt to your environment.

intelligent agent

See for instance [19], although we remember reading it in another source many years ago. We choose this definition over others because it provides a natural fit for agents: an *intelligent agent* is an agent that is able to adapt to its environment. In other words, as the environment changes an intelligent agent is able to adapt its behaviour to the new environment.

Arguably, this means just about any agent is an intelligent agent. Given an environment that can be in two possible states, an agent that is able to adapt its behaviour to both states is intelligent. Well, yes. The definition relates intelligence to an environment, and for this very simple environment, being able to handle both states is thus intelligent. Nevertheless, we are most interested in intelligence in complex environments with a very large number of possible states. An intelligent agent is then an agent that is able to adapt to a large number of states of its environment.

knowledge representation

One thing that also pops up once you talk about intelligent agents is *knowledge representation*. That is, the perceptions, the state, and the actions of an agent are all represented in a common format, which allows the agent to reason about them in a structured and presumably ‘intelligent’ way. This common format might be something like first order predicate logic, or a set of real numbers between 0 and 1. This also means the terminology changes slightly. Instead of saying that the agent holds data, its holds knowledge.

3.1 Rationality

Rationality is an important aspect of intelligence, although not the only one. Emotion for instance, is another[5]. Nevertheless, in artificial intelligence literature the terms are often treated synonymous [1][2]. The advantage is that rationality is much easier to define.

rational agent

A *rational agent* has preferences about the state of its environment, and it chooses action sequences that maximise the agents expected change of the environment according to the preferences. So if the agent prefers the environment to be in state X, a rational agent will choose an action sequence that maximises the chance that the environment will be in state X afterwards.

rationality

What is the rational thing to do depends on four things:[2]

- The preferences of the agent
- What the agent has perceived so far.
- What the agent knows about its environment. This is a combination of what its sensors perceive and what the agent deduces from those percepts.
- What actions the agent is able to take.

And given these four things, an ideal rational agent thus chooses the actions with the best expected result. Note that we keep using ‘expected’ and ‘chance’. We always assume an agent has an imperfect view of its environment, and hence it cannot be certain if its actions always have the intended result. However, an action that doesn’t have the intended result does not make the decision to take the action any less rational.

The last three points also give some nice pointers of how to make a perfect rational agent more intelligent: Giving it more knowledge, equipping it with better sensors, additional sensors, or better deduction skills, and enhancing its possible actions. All these result in an agent that is better able to adapt to its environment.

3.2 Beliefs, desires, intentions

A very interesting way to model the behaviour of a rational agent is using Beliefs, Desires and Intentions (BDI) [4][5]. We will try to explain this pretty complex matter.

A BDI agent’s state is differentiated in beliefs (what the agent knows), desires (what the agent wants), and intentions (how the agent intends to satisfy its desires). The beliefs, desires and intentions are all concepts stored in the memory of the agent, and we explain what they are next.

[5] Our Vulcan friends will tell you however, that emotion is overrated.

The reasoning of the agent is made up of four different processes: belief revision, situation recognition, planning and activation.

Figure 4 shows the basic structure of an agent brain based on this model, fitted to our previous model of the inside of an agent. Note that the heart is omitted from this picture. This is because the current picture resembles a single beat of a BDI brain. An agent built using BDI thus continuously executes this beat.

Also, note that we explicitly are dealing with temporal matters here. The memory contains historic beliefs, desires and intentions, which are updated to current beliefs, desires and intentions by the processes shown in the picture. At the start of the next beat, those current beliefs, desires and intentions become the new historic ones.

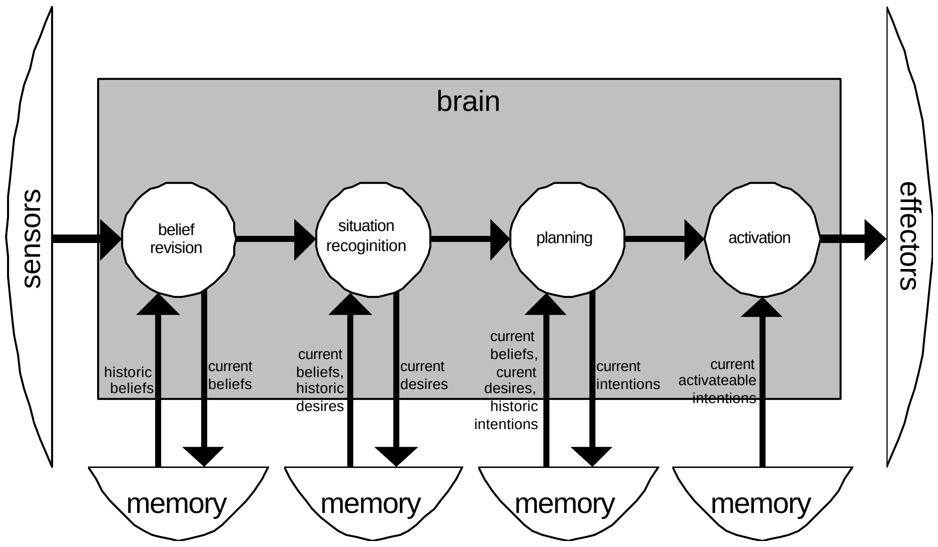


Figure 4: The process of beliefs, desires and intentions (one beat).

Beliefs

First, beliefs. Beliefs are things the agent knows, things the agent believes to be true. Beliefs are what is in the memory of the agent, together with the input from the sensors. The historic beliefs are updated, using the sensor input, by a process called belief revision. As the environment changes, so do the beliefs of the agent about that environment.

beliefs over facts

We don’t want to get too existential here, but ‘beliefs’ is used in favour of ‘facts’ because the agent is thought to have an imperfect view of its environment. It just doesn’t know or perceive everything. The environment might in fact be different from what the agent believes it to be and there might be other agents in the same environment that believe different, possibly even contradicting facts. To illustrate this, let us describe a scene from an episode of Babylon 5 [6]:

> A man has just woken up in an interrogation cell. The only light comes from lights in the ceiling. An interrogator walks into the room, shrouded by the light of an early morning sun. He asks the prisoner a few questions and leaves again. Sometime later he returns and starts with 'Good Morning'.
> The prisoner answers: 'It's not morning.'
> 'How do you know?'
> 'There was no light shining through the door when you came in'.
> 'Ah'.
> Subsequently, the interrogator gets up and signals someone on the other side of the door. We hear a click and the sunlight appears again.

Here we saw two different entities that both perceived the same environment, yet had different beliefs about that environment. The prisoner believed it was a certain time of day based on the light. The interrogator knew better. Of course, this exercise was deliberate, used to mentally abuse the prisoner.

Desires
On to desires. Desires *are preferred future beliefs*. A desire is satisfied once the current set of beliefs includes that preferred belief. So for a single desire, you prefer a future where you believe that desire is satisfied.

situation recognition

Historic desires are updated by a process called *situation recognition*, using the current beliefs as input. Example: Suppose your current beliefs tell you there is a shortage of nutrients in your body. Assessing this situation, you desire to eat. Once you eat some time in the future, the desire is no longer a desire, but a belief.

It is interesting to note that we humans use the temporal form of a verb to denote the difference between a desire (and later on, an intention) and a belief. As in 'I have eaten' (belief) and 'I want to eat' (desire). This makes it a bit difficult to see a desire as a future belief. We can do it by saying 'I desire to 'have eaten''. Then, once you have eaten, you believe you 'have eaten'.

Intentions

planning

Finally, intentions. Given a certain desire, you need to devise a plan that, given the current beliefs, will enable you to satisfy that desire. Historic intentions are updated to current intentions by a process called *planning*, which uses the current desires and beliefs as input. Suppose you desire to eat, and you currently believe you are in the living room, and there is bread and peanut butter in the kitchen. Then a straightforward plan would be: 'go to the kitchen' 'make a peanut butter sandwich', 'eat sandwich'.

You now see that there are several 'preferred future beliefs' added to your list of preferred future beliefs. These new preferred beliefs are your intentions. Intentions differ from desires in that the only reason you prefer them is to satisfy a desire. Also, they often are more concrete than desires.

You desire to eat, and to satisfy that desire you intent to eat a peanut butter sandwich.

plan

Intentions might be hierarchical, that is, each intentions might consist of several sub-intentions to achieve that intention. Intentions signify what the active plans are that you are pursuing, and the tree of intentions associated with a desire is the current *plan* for that desire.

activateable intentions

Ultimately, the leaf nodes of a plan tree are intentions that can be satisfied with a single activation of an effector. We will call these intentions *activateable intentions*. The planning process marks those activateable intentions that should be activated now in some way.

Activation

The final stage in this process, although not explicitly mentioned in the original articles, is *activation*. In activation, those intentions marked for activation are activated.

Now we have seen all the stages in the BDI process, let's see how this works out in an example. We start with the construction of a plan tree. This occurs in one BDI cycle.

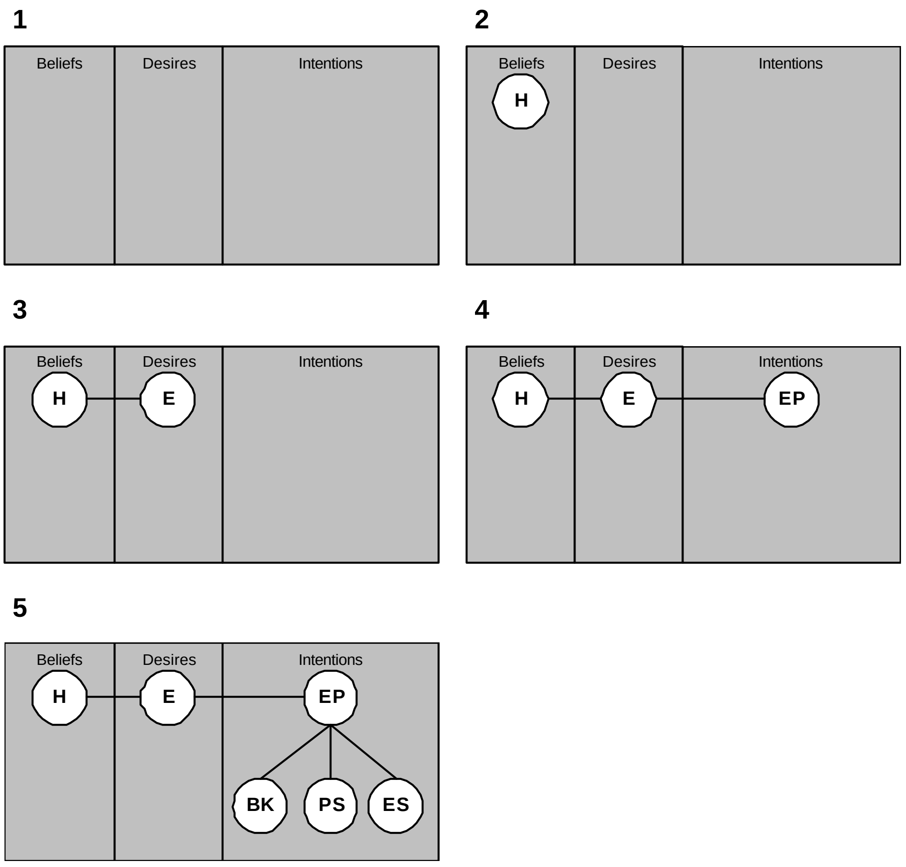


Figure 5: The construction of a plan tree.

Figure 5 depicts the peanut butter sandwich example again.

- **Step 1** shows the initial situation, with no beliefs desires or intentions relevant to our example.
- **Step 2** shows that after a belief revision, a new percept led to a new belief H, for Hungry.
- In s**tep 3** we see that this situation is recognised, resulting in a desire E for Eat.
- Next, in **step 4**, we have a top-level intention EP, for Eat Peanut butter sandwich. This plan was selected, presumably using other beliefs (not shown) about the current state of the environment, like that we are near a kitchen with a supply of bread and peanut butter.
- The last **step**, **5**, shows the plan fully fleshed out, with three sub-intentions: BK for Be in the Kitchen, PS for Prepare Sandwich, and ES for Eat Sandwich.

Assume for this example that BK, PS and ES are activateable intentions. In reality, there would be a huge tree of intentions, ending in a hideously complex set of intended muscle movements.

All these steps happened almost instantaneously, the immediate result of the belief to be hungry.

The next diagram shows how this plan is then executed, moving forward in time. Each of the steps illustrates the situation at the end of a cycle, right before activation.

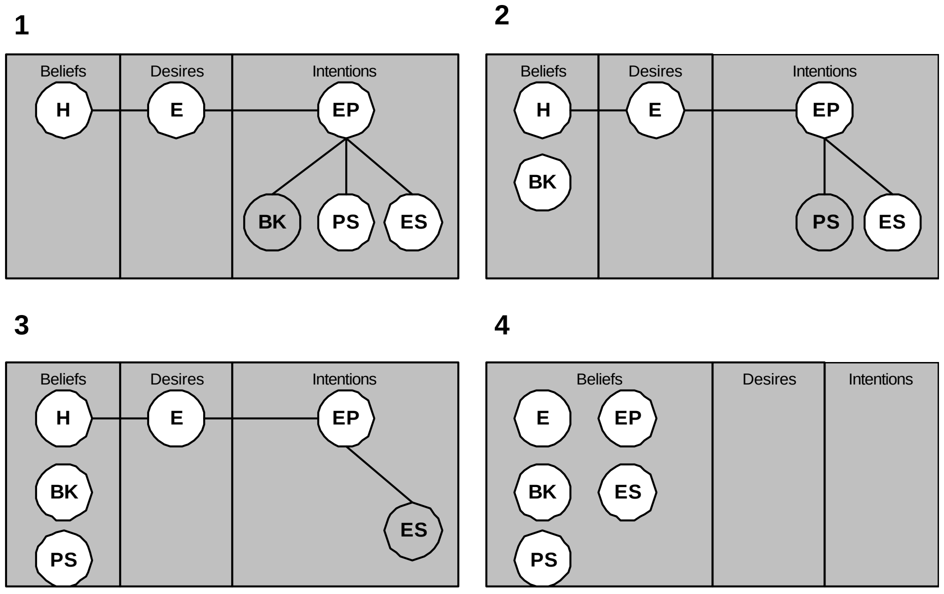


Figure 6: The execution of a plan

- **Step 1** is nearly identical to step 5 of the previous diagram. The only difference is that the intention BK is dashed, indicating it is to be activated next.
- **Step 2** shows that the walk to the kitchen was successful, and is now a belief.
- **Step 3** shows the preparation of the sandwich worked as well, and we now are going to eat.
- Finally, **step 4** shows the plan worked. We have eaten a peanut butter sandwich, and are no longer hungry.

Let's take a step back. What makes this model so different from 'normal' computation? After all, the plan tree looks a lot like a normal execution stack of a program. Well, the difference lies two things:

- The agent 'knows' what it is doing. It knows its own plans, where in normal code this is implicit.

- The three processes of belief revision, situation assessment, and planning. These allow the agent to adapt itself to a constantly changing environment.

Sure, a normal computer program might have some branching conditions at several stages in the program, but all possibilities have to be anticipated by the programmer beforehand. On the other hand, the BDI agent, by devising its own plans, effectively determines its own programming, which it can revise constantly as the environment changes.

3.2.1 A rational BDI agent

So how does one make a rational agent out of this? Well, recall rationality requires that the agent have preferences about the world and choose actions that maximise the preferences. For a BDI agent, this means the agent should choose plans that maximise the expected satisfaction of its desires.

This opens up a whole set of options as to how and what to maximise. For instance, in situation assessment, the agent might assign relative importance to desires. It might be interesting to quantify a desire, so for instance ‘hunger’ could gradually become more, so the need to satisfy it becomes more desirable. The planning process should thus try to choose a plan or plans that maximise the expected satisfaction of the current desires. Which plans that are could be highly dependent on the environment. Different plans might have an expected success rate associated with it, which might cause the agent to choose a ‘safe’ plan with a reasonable satisfaction over a ‘riskier’ plan with high satisfaction.

Also, time might be of importance. The agent might prefer a fast plan (with a small intention tree) to a slower plan, even if that slower plan gives a higher expected satisfaction of desires. Unfortunately, it is probably impossible to construct a perfect rational agent. Even with a few activateable intentions and limited beliefs the number of possible plans is very large, even infinite, if time is not an issue. The agent should probably be equipped with a set of plans with known expected future beliefs.

However, there are some interesting options. The agent could be given a desire of ‘curiosity’ that allow it to just experiment with activateable intentions in the absent of other desires. The agent could ‘learn’ from how these intentions influenced its environment and record the new ‘plan’ for future use. It could also try to revise old plans if a plan did not have the desired result.

3.2.2 Stacking BDI layers

BDI layers

One of the articles we refer to: *A pragmatic BDI Architecture* [4], actually has three BDI layers stacked on top of each other: The first layer is a reactive layer, which allows the agent to quickly respond to pressing situations. It is equipped with a limited set of situations that require immediate actions, and corresponding plans. Essentially, it allows the agent to have reflexes. The second layer, the local planning layer, works on higher level goals, and is able to devise plans that are more complex. The third layer is the co-operative planning layer, which allows it to co-ordinate its activities with other agents. We think this is a very good approach, and maybe it can even be generalised more:

BDI with feedback
We could introduce a feedback loop into the process described above.
It seems to us that such a construct could be used to create a very 'smart' agent, that is both able to respond quickly to critical situations, and able to think deeply should the need occur.

At the end of the planning stage, the agent might decide to continue 'thinking' or take action. The absence of a clear expected satisfaction of desires might be good indication to think a little further. Possible plans might have raised some questions about the environment that the belief revision process might be able to work on. This is very interesting because it can quickly become very laborious to perform all belief revision up front. To do the revision well, the revision process should do a lot of deduction, which might take a long time if there is no clear focus.

Consider the prisoner example earlier on. The prisoner might have only recorded the belief 'light shining through door' when the interrogator first entered the room. When on his second entry, the interrogator said 'Good morning' this created a desire to know the time of day. Planning might have been unable to create a correct response given a desire to 'answer truthfully'. Restarting deduction, the prisoner deduces that it probably isn't morning, since the earlier light seemed to indicate morning, and the absence of it now contradicts that. Therefore, the prisoner decided to contradict the interrogator.

Would the prisoner have been more paranoid, which we can see as a desire to know things for sure, the plan 'contradict the interrogator' might not suffice either. A new cycle, now given the fact that the belief it was morning was deduced indirectly, that contradicting the interrogator might be a bad idea, and that they might try to play mind tricks on him, might result in a plan to give a neutral answer.

3.3 Emotional agents

Computer programs that exhibit emotional behaviour are often deemed impossible. Indeed, a common theme in science fiction is the inability of robots to feel emotion, and what happens if they do. The core theme of the movie Blade Runner [8] is renegade androids that refuse to be deactivated and want to live a life of their own. Still, we have seen that it indeed is possible to create agents that seem to respond emotionally.

Lyotard

The article *An Architecture for Action, Emotion, and Social Behavior* [7] describes a system for emotional and social agents, centred on the construction of

Lyotard, a simulated cat. Lyotard is able to interact quite believable with a simulated world that includes other agents, most notably humans.

Lyotard’s actions in the world differ depending on its emotional state. For instance, if a human tries to pet Lyotard, and Lyotard dislikes that human, he will respond aggressively, by swatting or biting. If Lyotard likes the human, he will allow the petting, and feel happier afterwards. All this is accomplished by a system similar to that of a BDI agent described earlier. We describe some of the basic mechanisms using the BDI terminology, although in the reality the system is slightly different. Lyotard’s emotions are modelled as set of internal beliefs, such as fear, anger, happiness and love. These beliefs get their values depending on the environment the agent is in, probably somewhere at the end of belief revision or at the start of situation assessment. Lyotard will feel happy if his desires are satisfied, and sad when the satisfaction fails. Hope occurs when he perceives a desire is about to be satisfied, fear when a desire is threatened to fail. How much these emotions are affected depends on how important a desire is. Lyotard also associates a like or dislike with certain beliefs (which could be agents), depending on past observed beliefs. Lyotard likes a human that previously fed him and initially dislikes a new human.

Let’s quickly see how this results in the observed behaviour of Lyotard.

> Lyotard is in a room, and a new human tries to pet him. Lyotard thus perceives a new human, whom he dislikes. The human puts his hand towards him, which, because of the dislike, threatens the ever active desire of ‘not to be hurt’. This induces strong fear in Lyotard, which, results in anger towards the human, and thus desires ‘not to fear’ and ‘revenge’. The planning process now has to select a plan that will help satisfy those desires, and it selects a plan to ‘swat the human’ and ‘run away.’ The successful execution of this plan results in a new environment in a new room. This satisfies the desires ‘not to be hurt’, ‘not to fear’, and ‘revenge’, causing its emotions to return to neutral again, although Lyotard will still not feel happy.

3.4 Adaptive agents

adaptive agent

An intelligent agent that is additionally able to adapt itself in response to a changing environment is called an *adaptive*, or learning agent [1]. The difference is thus that an intelligent, non-adaptive agent responds the same way each time a given environment is encountered. An adaptive agent might respond differently, depending on its past experience. Also, an intelligent, non-adaptive agent might be baffled in the face of a new situation, whereas an adaptive agent should be able to devise a new approach. Or consider Lyotard the cat again. Our distrustful cat might gradually learn that most new humans do him no harm when they try to touch him, and he gets much more attention and food if he doesn’t swat them on the first encounter.

We examine adaptive agents by again using the BDI formalism. This is one of the reasons we like BDI so much: it is a very strong model for examining many different aspects of intelligent agents.

plan revision

A non-adaptive BDI agent has a fixed set of plans, and it combines those in intelligent ways to satisfy its desires. An adaptive BDI agent is able to create entirely new plans, and thus change its own behaviour. To do so, the belief revision process should be enhanced to include a *plan revision*, or learning step. The AI literature [2], but also psychology and other social sciences, recognise several different ways of learning. We examine a few of the more common ones:

Learning by example
Suppose the agent perceives a change X in his environment, and another change Y sometime later. This means there might by a cause-and-effect relation between X and Y. If the agent records this observation and the rest of the state of the environment in its memory, it could use this 'plan' at some time later if it desires Y and is able to do X (either directly or by another plan).

desperation

Learning by doing
This is a slight variation of learning by example. Instead of just observing changes in the environment, the agent itself takes some action and watches what effect that action has on its environment. In a simple case, the agent could 'randomly' try some actions to see what the results are. This could make sense if there is no plan available that could satisfy a certain desire. In effect, the agent acts out of desperation.

A slightly more advanced version would be if the agent *has* a plan that could satisfy a certain desire, but not all the conditions required by the plan are met. This might for instance be a plan it got by learning from an example. The agent could try the plan anyway, and see if it works or not. If it does, it knows the plan works in the current environment as well, and it could use some kind of inference process to adjust the plans conditions, like generalising some conditions or removing contradicting conditions.

Teaching
The first two cases assumed the agent was on its own. But now suppose the agent has other agents around it. (This makes this a case of social agents, see 2.8.3) Let's say we have two agents: a student, and a teacher. The student agent learns from the teacher. Either party can initiate the learning process. If the teacher initiates the learning, we get a model similar to a school system. The student listens to the teacher, and assumes what it learns comes in handy at some point. We can also have the student ask a question, so ask for a way to satisfy some desire.

The actual learning can roughly take two forms: If the teacher influences the environment, and the student observes the effects of this, we get another case of learning by example. We can also have the teacher give the student a new plan directly. In both cases, a new plan gets stored in the agent's plan library.

3.5 Conclusion

Intelligent agents are one of the most important and well-known types of agents. We say an *intelligent agent* is an agent that is able to *adapt* to its environment

To be able to reason about all it has perceived so far, an intelligent agent uses a uniform way of representing this information (resulting from percepts), called a *knowledge representation*.

A rational (intelligent) agent is an agent that has some preferences about the future state of its environment, and takes the best (for some definition of best) actions to get its environment into preferred states.

A very interesting technique is for creating intelligent agents is BDI: *Beliefs*, *Desires* and *Intentions*. Together, these three sets of items form the state of the agent.

The beliefs are what the agent has perceived so far, they represent the current and historic state of the environment to the best of the agent's knowledge. The beliefs of the agent are updated by a process called *belief revision*, which updates the set of beliefs based on the current perceptions.

After this, the agent performs a *situation recognition* step, resulting in set of desires about the future state of the environment.

The next step is *planning*, in which the agent creates a *plan*, a tree of intentions, for each desire the agent has. The leafs of each plan are *activateable intentions*, intentions for which the agent knows it can take actions that will make that intention happen.

The last and final step is then *activation*, in which all the actions corresponding to all the activateable intentions are taken. Once the environment is in the intended state, the intention becomes a belief, and the execution of the plan continues.

An *adaptive* agent differs from an 'ordinary' intelligent agent in that it is able to acquire and use entirely new plans. How it obtains these plans can take different forms, such as *learning by example*, *learning by doing*, or *teaching*.

CHAPTER 4 Modelling and agents

It is by now commonly understood that for all but the smallest applications, you need to go through a thorough analysis and design phase prior to writing code. An application often starts with a rough idea of what the application is supposed to do. During analysis, this rough idea is fleshed out into a thorough description of the functionality and requirements of the application. A subsequent design then describes how the application should be implemented. Research proves that doing solid analysis and design prior to coding substantially increases the odds that the resulting application meets the requirements, and it also improves the stability and maintainability of the code.

All this does not mean however, that you cannot go back to a previous step when already developing. It is very normal that during design you find that something was missed in the analysis phase, which you then subsequently update. In today's fast paced world, it is often impossible to anticipate all the requirements up front. It might also happen that during the development of the system external factors change, calling for a change of some part(s) of the system. When developing a large scale system, it is often better to do a global analysis of the system, then design some core parts of the system, code those, and then go back to designing other parts of the system.

The propaganda above describes why you should do analysis and design prior to coding. You might be wondering why we wish to discuss this in this thesis. The reason for this is twofold: first, it is not immediately evident how to model an agent based system, and we want to show a possible way it can be done. The second reason is that, during the analysis and design of a 'normal' application, you might encounter hints that point to agent-like behaviour of parts of the system. It could be important to look for those hints, because we believe that modelling those parts as agents could make designing the application easier.

Our modelling method is an enhancement of UML, the Universal Modelling Language. UML in its current from is fairly new, so we quickly introduce this modelling method for readers that might not be familiar with it, and to make sure readers understand our notation conventions.

After this we introduce our enhancements to this modelling methods, and in the process show what an agent based design looks like in (our version of) UML. The most important addition is the distinction between animate and inanimate objects, which allows us to model which pieces of a system can act autonomously. Although this thesis is about distributed applications and agents, we defer modelling distributed applications to chapter 4.

4.1 UML

model

There exist many different ways, or methodologies, to do analysis and design. One of the common and well known ones is the Universal Modelling Language (UML) and the accompanying methodology. UML, like many other methodologies, lets you first develop a *model* of the system prior to developing it. A model, to quote the well-known Dutch economist Heertje, ‘is always less than reality, except for a super model, which is more than reality’.

A model thus only shows the essential parts, providing the very important ‘big picture’. Because nothing is built yet, a model is much easier to change. As implied here, UML is generally used to model software application. This is not mandatory however. The most important aspect is that it allows you to capture and analyse certain aspects of reality.

UML supplies a set of primitives to construct a number of different diagrams, and the UML methodology describes how and when to use the different diagrams. Most diagrams are generally accompanied by a textual description that further clarify the elements in the diagram. Together, the diagrams and descriptions make up a *UML model*. The good thing about a UML model compared to some other methodologies is that it is very resilient to change. Even if functionality changes dramatically, the effect on the model is often little more than a few changes in some of the diagrams and descriptions.

UML was created by ‘the three amigos’: Rumbaugh, Jacobson and Booch, and is basically a nice, consistent union of each of the contributors own modelling methods. UML is object oriented (OO), meaning it is based on the simple idea that basically everything around us can be seen as an object, and so it makes sense that if we want to model a system, we model it using familiar objects as encountered in real life. For this reason, a UML model is sometimes also referred to as an *object model*.

One of the nice aspects of UML is that it that the same model can be used throughout the lifetime of the application, from the first analysis all the way through maintenance. The model is of course refined and changed as time progresses, but you don’t have to switch to a new model once you enter a different stage.

Even though UML is by far the best modelling language we know, it has some gaps as it comes to modelling with agents. We think this is because UML is grown from a base that largely focussed around the creation of user-centred but otherwise passive applications. Luckily UML defines an extension mechanism to be able to tailor it to our specific needs. We have tried to keep this extension as minimal as possible, because we do not believe the world would be helped by us unnecessarily cluttering UML.

We briefly describe some of the methodology and diagrams but for a more thorough description of it we refer to [34] or to one of the tutorials on the subject, such as [35].

4.2 A crash course in OO terminology

As said, object orientation means thinking about the world in terms of objects, and to do object oriented modelling means you model a system and its behaviour by partitioning it in a set of interrelated objects, each with specific qualities. Doing so provides a very useful abstraction away from specific code or functionality.

4.2.1 State and behaviour

State, behaviour

Formally, an object is characterised by *state* and *behaviour*. State means an object has particular qualities, as a book has a title, an ISBN number, contents, and a cover illustration. State can change over time, for instance the pages at which a book is open can differ. Behaviour means the object can be interacted with in a certain way, such as a book can be moved or opened.

4.2.2 Class

class

Objects are thought of as belonging to a particular *class*. A class is a group name for similar objects.

instance

For instance, ‘Book’ can be a class, and most books have a title, and ISBN number and the other qualities we mentioned above. An object of a particular class is said to be an *instance* of that class, so ‘Lord of the Rings’ is an instance of the class ‘Book’. When modelling, it is usually the class of an object that is the most important, it determines how the system ‘sees’ the object.

4.2.3 Methods and attributes

attribute, method

The state of an object is made up of zero or more named *attributes*, behaviour is made up of zero or more named *methods*. Objects of the same class share the same attributes and methods, but the value and of those attributes and the exact behaviour of each of the methods might vary.

So for a book, we can have attributes ‘title’ and ‘ISBN number’, and methods ‘move’ and ‘open’. ‘Lord of the Rings’ and ‘The Hitchhiker’s Guide to the Galaxy’ both are books, but their attributes differ.

parameter, return value

Methods have zero or more *parameters*, which influence the behaviour of that method. The ‘read’ method might have parameters ‘page’ and ‘line’, that determines which line you want to read. Methods also have a *return value*, the result of that method. A method 'getTitle' on a book would return the value of the title attribute.

signature

The name, parameters and return value of a method together are called the *signature* of that method. A signature of a method is written by the name of the method following a comma-separated list of parameters in parenthesis, like this: 'read(page,line)'

These few paragraphs quickly mentioned the basic principles of object orientation. But OO also includes a few more advanced concepts, which we will try to describe next.

4.2.4 Abstraction

abstraction

Even though a real life object has many different attributes and methods, you should only use those that are relevant to your model. This is called *abstraction*. Abstracting away unnecessary details allows you to better focus on the problem at hand. For instance, the text in a book is printed in a particular font. If you are not the publisher of the book, then this is probably not that relevant to you and so is not part of your model of the book.

4.2.5 Encapsulation

encapsulation

How an object implements its behaviour or represents its attributes is only of concern of the object itself. The object *encapsulates* its own state and behaviour. This allows you to treat the object as a 'black box' if you are not directly focussing on it.

4.2.6 Inheritance

inheritance

Inheritance comes into play when you think of objects belonging to a hierarchy of classes. For instance, our sun belongs to the class 'Star'. But it also belongs to the more general class of 'celestial bodies'. Planets and asteroids are examples of other classes that also belong to this same, more general, class. The general class is called the baseclass or super class, the specific class is called subclass or derived class. Inheritance states that a subclass inherits the methods and attributes of its baseclass. The subclass can introduce additional attributes or methods.

Suppose the class 'Celestial Body' has one attribute: 'position', modelling the position of the body in the universe. Each object of class 'Star', 'Planet' or 'Asteroid' thus inherits this attribute. A planet and an asteroid might include a method 'move' to update their position in the universe, and a planet might have an additional attribute describing which star it belongs to.

abstract class

In this example, there will never be an object that is just a celestial body. All the objects belonging to that class will always be an instance of some sub class. Such a class is called an *abstract (base)class*. It exists only to be able to generalise certain qualities. An abstract class is allowed to have one ore more methods that are abstract as well, meaning the class does not provide an implementation for it.

4.2.7 Polymorphism

polymorphism

Polymorphism is a direct result of inheritance. *Polymorphism* means 'being able to take more than one form', which in OO translates to 'able to be more than one class.' Any class that is a subclass of some base class is able to be that base class as well. It inherited the methods of the base class, but a sub class can choose to behave differently if it wants to.

Getting back to our celestial body example, suppose we abstract away the fact that stars don't really move, so we can add a method 'move' to the 'Celestial Body' class. This allows us to have a simple universe with moving objects by periodically calling the 'move' method of each class. Each of the classes 'Star', 'Planet' and 'Asteroid' behave differently for this method. The star doesn't move at all, the planet updates its position in relation to its star, and the asteroid updates its position based on its current speed, direction and proximity of other celestial bodies.

4.2.8 Association

association

Two classes can be *associated* with another, as the planet was associated with a star in the previous example. This association can have a name, like 'is the sun of' for stars and planets.

multiplicity

It is often important to know how many instances of the each of the classes participate in the association. For instance, a particular star might be the sun of three planets. In general, a star is the sun of zero or more planets. The possible number of instances of each side of an association is called the *multiplicity* of that side. For the 'is the sun of' relation, this means the multiplicity of the star side is one, and that of the planet side is 'zero or more'. When speaking about the association, you put the least multiplicity first, and connect them by 'to'. So, 'A star is the sun of 1 to zero or more planets' This might seem a bit convoluted at first, but it is very precise.

aggregation

Aggregation is a stronger form of association. You speak of aggregation when an object is part of another object, as an atmosphere is part of a planet.

4.3 Use cases

Now, with the OO terminology in our back pocket we can start analysing the system at hand. After the initial brainstorming on the application, analysis starts with developing use cases.

use case

A *use case* is a textual description of a possible interaction with the system. Each use case has a name, a short sentence describing what is accomplished in the use case. For a word processor application, examples might be 'open file', 'enter text' or 'check spelling'. All use cases of a system should together describe all the functionality of a system.

actor

A use case is initiated by an *actor*, an entity (person or other system) outside of the system to be modelled. Each of the steps in a use case is generally considered to execute synchronously, that is, the actor or the system waits while the other party executes the step.

If a step is inherently lengthy or might take a widely varying amount of time, the continuation is a best modelled as a separate use case. In the mean time, the actor can go off and do other things, so either initiate another use case or not interact with the system at all. At a later point in time, the actor then initiates the continuation use case, which would probably include a check at the beginning that the lengthy operation is completed.

Suppose you are modelling a book selling web site like the famous Amazon.com [30]. For a customer of the web site, a possible use case is 'look up information on book' another is 'buy book'. The second one might look something like this:

> **Buy Book**
> Customer selects book to buy.
> System displays a payment and shipping options.
> Customer selects payment and shipping option, and fills in payment details.
> System displays a total order estimate and asks for confirmation.
> Customer confirms order.
> System verifies clients credit and displays order confirmation.

For this use case, the actor is thus 'Customer', who goes through a set of steps with the result he or she has bought a book. Here, the Customer and the System take turns, but this need not be the case. It might very well be the actor does something simple, and the system goes trough a whole list of complex steps.

The description above reflects various choices made for the system. If you add the concept of a shopping basket, you need create two other, different, use cases: 'adds book to shopping basket', and 'buy items in shopping basket'.

The description of this use case is still somewhat rough and for instance doesn't include what happens if the selected book is out of stock or the customer has insufficient credit. Still, it is quite adequate for a first cut.

system use case

Use cases are not restricted to 'target audience' use cases. Equally important, but often forgotten, are system use cases, those of employees maintaining the system, such as 'add new book'.

4.3.1 Use case diagram

use case diagram

Use cases come with a diagram, the surprise: *use case diagram*. A use case diagram usually shows more than one use case. The diagram consists of stick figures, the actors, and ovals containing the names of the use cases. Arrows show which actors initiate which use case. For our book web site example, this looks like this:

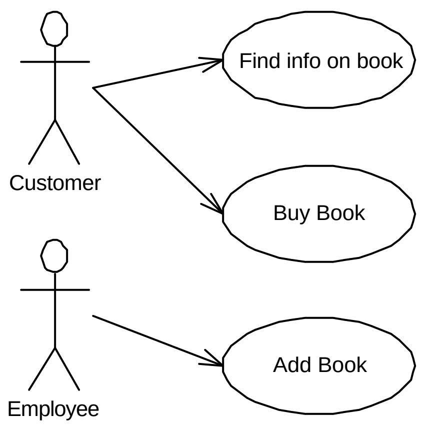


Figure 7: Use case diagram for book web site.

Here we see two actors: ‘Customer’, ‘Employee’, and three use cases, ‘Find info on book’, ‘Buy book’ and ‘Add book’. The customer initiates the first two use cases, the other the last.

A use case diagram of all the use cases for a system gives a quick overview of the functionality of a system.

4.3.2 Conditional statements and inclusion

conditional statements

Suppose we do need to describe what happens if the book is out of stock. For this, we need to include branching points in the use case, points where the next steps differs depending on some condition. This is done by introducing *conditional statements* into the use case.

if, while

There are two types of conditional statements: IF and WHILE. We print both in capitals to clearly distinguish them form the rest of the text. Both statements are followed by some condition, and the steps after the conditional statements are only followed if the condition is true.

The difference between the two is that for an IF, the steps after the statement are only followed once, while those after a WHILE are followed as long as the condition holds. The steps after an IF can be followed by an optional ELSE statement, which then means those steps are followed if the condition doesn’t hold.

It is good practice to indent the steps after a conditional that should only be executed if the condition holds. Now a second version of ‘buy book’ this time with conditional statements:

Buy Book

```
Customer selects book to buy.
IF the book is in stock
    System displays payment and shipping options.
    Customer selects payment and shipping option, and fills in payment details.
    System displays a total order estimate and asks for confirmation.
    Customer confirms order.
    System verifies clients credit
    IF the customer has enough credit
      System displays order confirmation.
    ELSE
      System informs the customer he or she does not have enough credit
ELSE
    System informs the customer the book is out of stock
```

This use case description thus has two branching points, and as a result three possible outcomes.

include

Now, use case inclusion: a use case can include another use case. Inclusion is indicated by putting in INCLUDE, followed the name of the other use case. In a use case diagram, inclusion is indicated by an arrow pointing to the included use case. Including another use case is useful to avoid repeating an identical set of steps in different use cases. The common steps are put in a separate use case, and all use cases that need those steps include that use case. Let's allow for the purchase of CD's as well. Then all logic after an item is selected in the same:

Buy Book

```
Customer selects book to buy.
INCLUDE Show payment options
```

Buy CD

```
Customer selects CD to buy.
INCLUDE Show payment options
```

Show payment options

```
IF the item is in stock
    System displays payment and shipping options.
    Customer selects payment and shipping option, and fills in payment details.
    System displays a total order estimate and asks for confirmation.
    Customer confirms order.
    System verifies clients credit
    IF the customer has enough credit
      System displays order confirmation.
    ELSE
      System informs the customer he or she does not have enough credit
ELSE
    System informs the customer the item is out of stock
```

And the accompanying use case diagram:

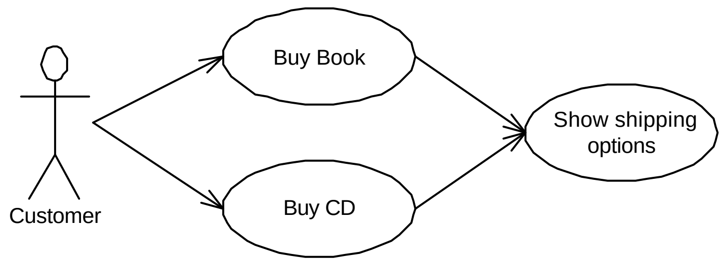


Figure 8: Buying books and CDs.

4.4 Class Diagram

class diagram

The *class diagram* is the most fundamental element of the UML methodology. It shows which classes together make up the system, and how those classes relate to each other. Optionally, the class description can contain the methods and attributes of each of the classes. Larger applications usually have multiple class diagrams, which for instance group the classes by functionality or subsystem. There is nothing wrong with including the same class on multiple diagrams, although generally only one occurrence would have the methods and attributes of the class.

Once the use cases are at a reasonable stage, it is good to start on the class diagrams. Creating a good class diagram takes practice, but a good head start is to find all nouns mentioned in the use cases. Each of those nouns could become a class on one of the class diagrams.

Examining the ‘buy book’ use case described earlier, we find the following nouns: customer, book, payment option, shipping option, payment details, System, total order estimate, order confirmation, order, credit. This gives rise to the following class diagram:

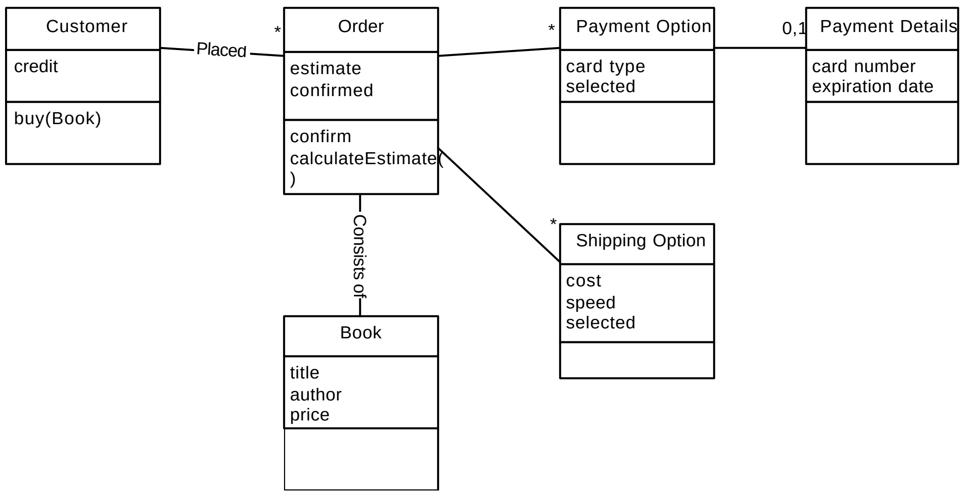


Figure 9: Class diagram for book web site example.

This diagram shows several of the aspects mentioned in 4.2. There are six classes, each represented in a rectangle. Each rectangle is subdivided in three sections: The first shows the class name, the second the attributes of that class and the third has the methods. The lines between the rectangles show the associations between the classes. The symbols at the ends of the lines indicate the multiplicity of the relation. No symbol indicates ‘1’, a star means ‘zero or more’ (‘many’), and 0,1 means ‘zero or one’.

This diagram is by far not the only class diagram possible for the book web site, and reflects several choices we made. For instance, ‘order estimate’ and ‘order confirmation’ are not made into separate classes, but are considered attributes of the ‘order’ class. Likewise, ‘credit’ is an attribute of ‘Customer’. It is fairly common not to make ‘system’ into a class. You can think of all the classes together forming the ‘system’ class. Or, in OO terms, the system class is the aggregate of all the other classes.

Note that there are some attributes on the classes that do not directly appear in the use cases. We added them to make the diagram a little more interesting. You can imagine however, that if a ‘shipping option’ is elaborated a little more in the ‘Buy Book’ use case, you need to know how fast it is (i.e. three to two five working days’, or ‘overnight service’) and what the costs of the option are.

Notice that the association between ‘Order’ and ‘Book’ is one to one. Showing this to the client, he or she might say: ’Hey, but we want the customer to be able to buy more than one book at a time.’ At which time you could reply ‘Ah, but then we need to change the functionality, and allow the user to add items to a shopping basket.’ Changing this functionality creates two very different use cases. The effect on the class diagram would be that the association between ‘Order’ and ‘Book’ changes to ‘1 to many’, and maybe add an ‘add(Book)’ method to the order object. This is a nice example of how a big change in functionality has only minor impact on the object model.

4.4.1 Inheritance and aggregation

Inheritance and aggregation are two of the concepts of 4.2 that we did not show yet. As you might recall, polymorphism is a result of inheritance, and you see it in this diagram:

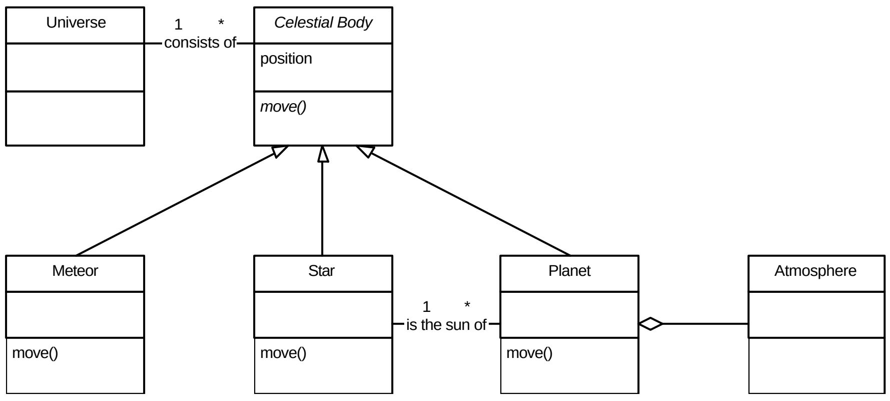


Figure 10: Universe class diagram.

This diagram shows a couple of new elements compared to the previous one. Let's start with the Celestial Body class. Notice that its name is written in Italics. This is to indicate the class is an abstract class. The method name in Italics means that the method is also abstract. The three classes Meteor, Star and Planet are all subclasses of the Celestial Body class, shown by the triangle at the end of the connecting lines. Because these have to be real (not abstract) classes, they all provide an implementation of 'move()', as shown.

The line connecting Atmosphere to Planet has a diamond on the end, indicating an atmosphere is an integral part of a planet, or, Planet is an aggregate of an Atmosphere and possibly other things (not shown). Finally, notice that Universe is associated with zero or more Celestial Bodies. This is an example of polymorphism. In reality, each Celestial Body is an instance of one of the three sub classes. However, from the viewpoint of the Universe, they're all Celestial Bodies.

4.4.2 Stereotypes

stereotype

UML has a standard mechanism for categorising classes, called *stereotyping*. By stereotyping a class, you indicate it belongs to a particular category and/or has a special property. Stereotyping a class is similar to subclassing it, but sort of on another dimension. Two classes that have the same stereotype might not share any attributes or methods, and vice versa, a subclass does not necessarily 'inherit' a stereotype of its base class.

An example stereotype is 'singleton'. A class is singleton if there will never be more than one instance of that class in a system. Another example is 'interface'. A class is an interface if it is abstract and does not have any attributes.

Interfaces are an incredibly useful construct. An interface is like a contract between two classes. A class using an interface just sees that interface, it is not aware of the specific class that implements (is a subclass of) the interface.

In a class diagram, a stereotype is indicated by the stereotype name in guillemets («,») below the class name, like this:

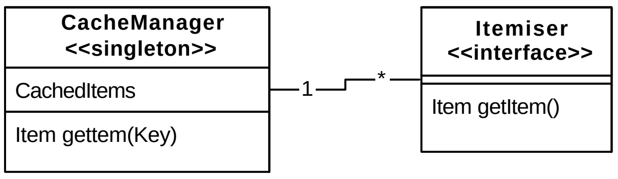


Figure 11: stereotyped class.

The picture shows the 'CacheManager' class, which in this case holds some sort of items. A cache is a typical example of an object that should only have one instance in a system.

The 'Itemiser' presumably uses the cache to get some item instead of retrieving them from some other source. There could be many Itemisers in the system. They would all be a different class, but they would have the same interface.

4.5 Sequence diagram

sequence diagram

While an object diagram shows how objects relate to each other statically (in space, if you will), a *sequence diagram* shows how several objects work together in time to accomplish something.

A very frequent use of a sequence diagram is to show how a use case works out once all the classes are defined. If the sequence diagrams can be created for all the use cases, this is a good indication the class diagrams are complete. However, it might still be advantageous to create additional classes to split off functionality etc.

Let's make a sequence diagram for the buy book use case (4.3.2), including all the shipment and payment option selection:

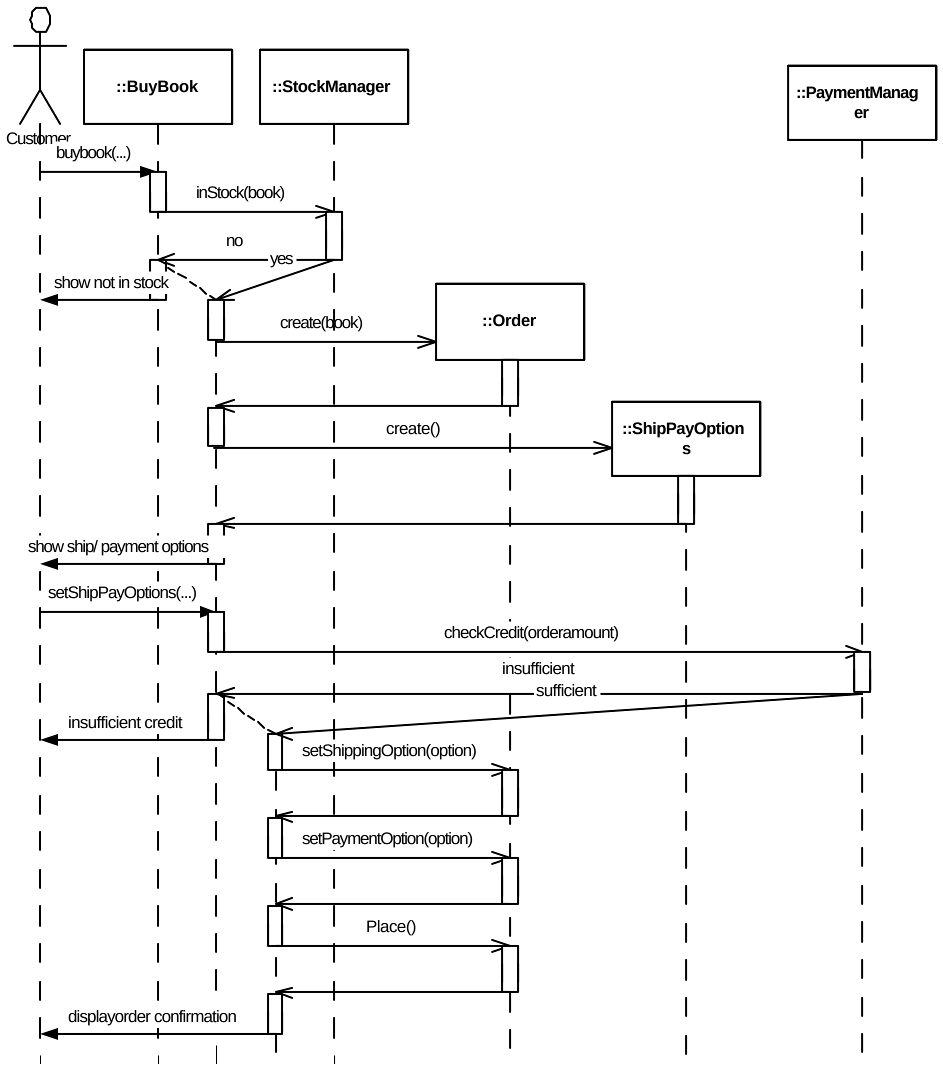


Figure 12: sequence diagram for 'Buy book'.

This diagram might look pretty overwhelming at first. We will try to explain all the subtleties that are in this diagram. The dashed vertical lines are object lifelines. On top of the line is the name of the class (object) that is represented on the lifeline. Time passes from top to bottom.

activation

Most of the object lifelines have small filled rectangles superimposed on them. Those are called object *activations*. They mean that, at that moment in time, that object is activated (doing something). Please note that the length of the activation is not an indication of the duration of the activation.

message

The arrows back and forth between the lines are *messages*. In this case, an arrow from left to right is a method invocation, and an arrow from right to left the return of that method call.

split

You see the lifeline for the 'BuyBook' object splits two times. Each split means there are two possible sequences from that point on, and which is followed depends on some condition. In this case, the branches correspond to the branches in the use case description. The lines might join up again if the sequence is the same for both paths after that.

With this explanation in mind, we hope you can follow the process of sending placing an order for a book. Note that there are few extra elements in this diagram that aren't on the class diagram: One new class, 'BuyBook' manages the different actions needed to successfully complete an order. We discovered we needed these items when we tried to draw this diagram. This is a nice example of how an object model evolves, and how the different diagrams influence each other. In a real design process, the next logical step would be to revise the class diagram, but we will just move on.

4.5.1 Message types

simple messages

The diagram above showed messages send back and forth, representing method invocations. These are *simple messages.* Pure model wise, you could have a simple message only one way. In practice however, a simple message is always one back-and-forth pair. When we say 'method invocation', we mean one such pair. The first message corresponds to the passing of the parameters of a method, the second to the return value (even if there is none) of the method being returned to the originating object.

Simple messages are not the only type of message possible however, there are two other types, synchronous and asynchronous.

synchronous message

For *synchronous*, the activated object stays active, but waits for the other object to return a message.

asynchronous messages

For *asynchronous messages*, the activated object also stays active, but doesn't wait for a return message, and there might be none.

Note that in both these two cases, there are two active objects at the same time, which is why it is not simple. An example might help to clarify the differences.

You, as a human are an actor, which means can send messages to other objects. Suppose you need to cut up some mushrooms. There are three options:

Simple
You take a knife and start cutting up the mushrooms yourself. You activate the knife, which cuts the mushrooms. This is simple messaging.

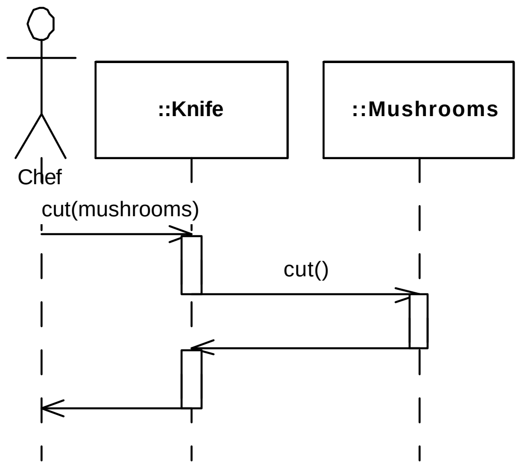


Figure 13: Cutting mushrooms with a knife.

Synchronous
You throw the mushrooms into the food processor, and press the button. You wait for the mushroom to be cut up, and take them out. Here you see you wait while the other object is doing something. This is synchronous messaging, you wait for the 'message' of the food processor that it is done cutting the mushrooms.

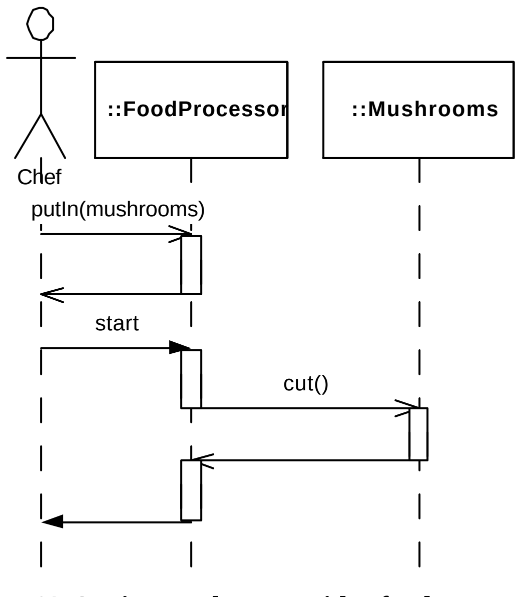


Figure 14: Cutting mushrooms with a foodprocessor.

Asynchronous
You again throw the mushrooms into the food processor, but now you start to peal some onions. If the food processor is done, and probably after you are done with the onions, you pay attention to the mushrooms again.

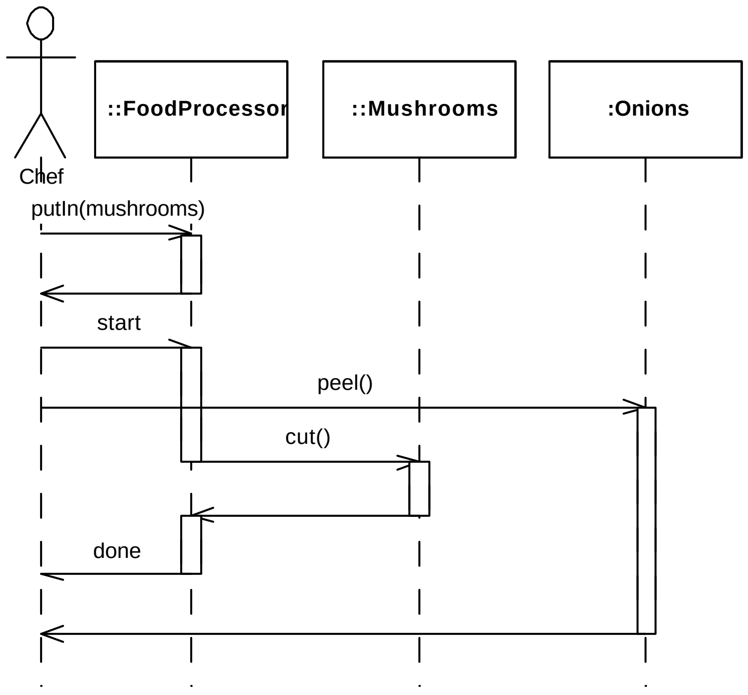


Figure 15: Peeling onions while the mushrooms are cut.

The advantage of asynchronous messaging is clear, the calling object can continue doing something useful while some other object is doing something for him as well. But you might be wondering what the need is for synchronous messaging. Well, in some cases it just isn't possible to transfer the activation to another object, for instance if the object lives on another machine. In that case the object on the sending machine has to wait until a reply arrives.

4.5.2 Object construction and destruction

The sequence diagrams we've shown so far assume all the objects involved in the interaction exist beforehand and continue to exist afterwards. Sometimes however, a new instance of an object is created, or and existing object is destroyed.

object creation

In a sequence diagram, *object creation* is shown by drawing the object box halfway during the interaction at the moment of creation.

object destruction

The *destruction* of an object is shown by ending the object's lifeline with an X.

The following sequence diagram illustrates both:

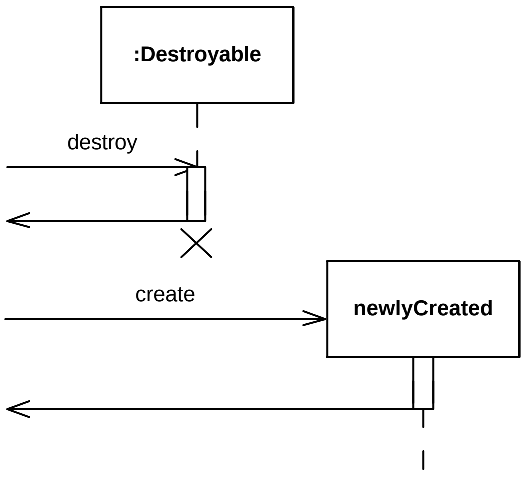


Figure 16: Destroying and creating an object.

4.6 State Diagram

state diagram

We already said in 4.2.1 that the state of an object can change. If an object can be in many states, and the behaviour of that object is different depending on its state, it is useful to create a *state diagram* for it.

state change

Note that in this context, a ‘state change’ does not mean just a change of the value of some attribute. However, a change of an attribute value could induce a state change. For instance, if the last money is withdrawn form a wallet the wallet goes from ‘containing money’ to ‘empty’. A subsequent attempt to withdraw money would obviously fail. This type of state changes is modelled by a state diagram. A state diagram always contains the state changes of a single object.

Let’s examine this by looking at the order class from the book web site example, with the extra assumption that we are using a shopping basket, and that in our system, shopping basket contents is modelled as an order.

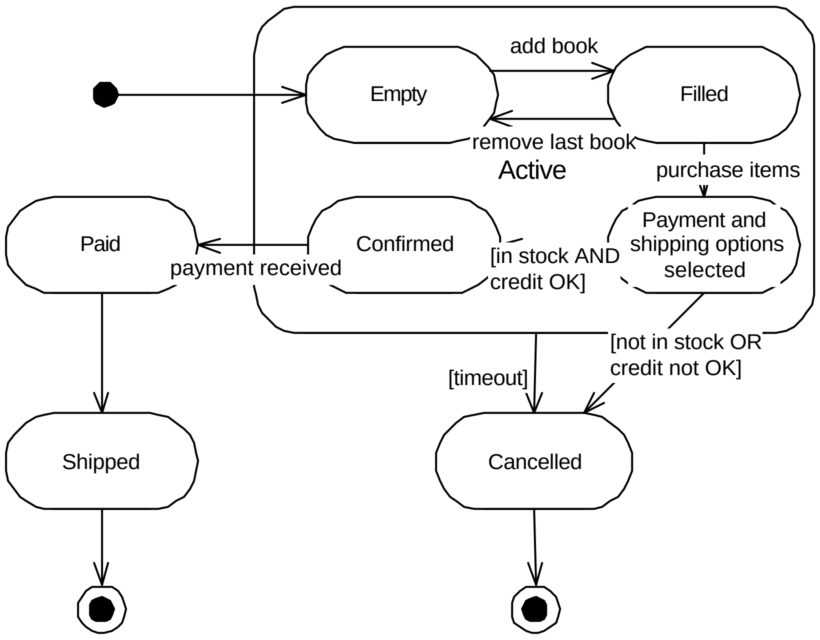


Figure 17: State diagram of book order

Impressive isn't it?[6] Well, let's examine the elements in this diagram.

state

A *state* is represented by a rounded rectangle, with the name written in the middle. An 'Order' object can be in four different states:
active, cancelled, paid and shipped.

initial state, end state

The filled circle is *the initial state*, pointing to the first state the object is in upon creation. The filled circle with outer ring is an *end state*, in which the object is destroyed.

state transition

Outgoing arrows indicate the possible *state transitions* from that state to another. So in our example, an object starts in the 'active' state, then moves to either 'Cancelled' or 'Paid'. From 'Paid', the object transitions into 'Shipped'.

substate

The 'Active' state is special, because it has four *substates*. When 'Active', an order can be 'Empty', 'Filled', 'Payment and options selected' or 'Confirmed'. Substates are convenient to show an object can transition to certain state from a set of other states. In this case, we want to show an order can become 'cancelled' from any of the four sub-states.

event

Many of the transition arrows are adorned with text descriptions. These descriptions show what *event* happens at the state transition. This might be an event that cause the transaction, or something that happens when the object enters the new state.

[6] That is, if you hadn't seen a state diagram before. Otherwise, you were probably thinking 'Huh, I've seen bigger'.

guard conditions

Three of the descriptions are written between square brackets ([,]), making them *guard conditions*. A guard condition is a condition that needs to be met for the transition to take place. So you see that for the 'payment and shipping options selected' state, whether the order is cancelled or confirmed depends on whether the books are in stock and the customers credit is sufficient.

4.7 Other diagrams

UML has a few other diagrams that we do not cover in this chapter, mainly because we do not expect to use them later on. Those are the collaboration diagram, the activity diagram, the component diagram, and the deployment diagram. For completeness, we briefly discuss what they are and they are used for below.

Collaboration diagram
A collaboration diagram is a variation on a sequence diagram. Whereas the sequence diagram emphasises time, the collaboration diagram shows the relations between the objects spatially. As such, it is somewhat clearer which objects interact with each other, and why. Because the time aspect is very important with agents, we chose to use the sequence diagram over this diagram.

Activity diagram
An activity diagram is an extended form of a state diagram, with explicit support for decision making, signalling between different objects, and concurrent paths. It is very useful for modelling business processes and other flows of control.

Component diagram
Unlike most of the other diagrams, this diagram shows real life entities, instead of abstract concepts. It shows which software components make up a particular system once it is build. Although this is extremely useful, we don't really have a need for this in this thesis.

Deployment diagram
Finally, the deployment diagram shows which hardware components a systems is made up of, and which software components are deployed on that hardware. Again, this is far to deep into deployment to be of interest to us.

4.8 Object oriented programming

Object oriented programming

We briefly want to discuss the relation between object oriented programming and object oriented modelling. *Object oriented programming* means you still treat a system as being composed of objects when implementing it.

It is not strictly necessary to use an object oriented programming language such as Java or C++ to do object oriented programming, but not doing so forces you to do some pretty heavy wizardry yourself. Especially a feature as polymorphism is pretty tough to implement using a non-OO programming language, which has that and other features built in.

Doing object oriented analysis and design (OOAD) and not using an object oriented programming to implement the system is downright silly. The other way around is a little less odd, and effectively means the programmer is building the object model on the fly, while programming. This can work, but you miss out greatly on the oversight and ease of change you get when doing OOAD first.

Apart from so called 4GL packages, most programming today is done using an object oriented programming language, in particular Java or C++, so we will focus on that. When showing code examples, we'll use Java as the language of choice, because of its wide use and because it is one of the few that has nice built-in support for multithreading, which we'll need to create animate objects (see chapter 6).

4.9 Using agents in modelling

As we said, UML is largely suited to model reactive systems, systems that respond immediately to outside stimuli. This is especially apparent with use cases. Use cases are supposed to cover all the functionality of a system, but, at the same time, use cases are initiated by actors, which are always outside the system. This means that if we have functionality that is initiated by the system itself, we have no apparent way of putting this functionality into a use case. As systems get larger and more proactive, this deficiency becomes increasingly apparent.

A real-life example:

> A system we recently developed allowed a user to create a list of interests on which he or she would like to receive email alerts. The system would monitor the incoming news, and send an email to the user if the topic of news item was included in that users interest list. The model for that system included a nice use case for the user updating his interest list, but a use case for sending out the emails was suspiciously missing. The reason for this was simple: no user or external system initiated the action, so the modellers were unsure how to put the functionality in a use case. For this system, this omission was handled because a description of this functionality appeared in a high-level functionality description, and because we basically kept reminding ourselves the system had to be capable of doing this.

4.9.1 The system actor

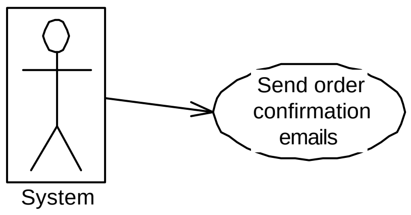
system actor

A simple solution to this dilemma would be to introduce a special *system actor*, one that we allow initiating system functionality. We could use the same symbol for this actor, but somehow the stick figure appearance of a normal actor doesn't feel quite right. We therefore introduce an additional element, the system actor. Its current depiction is a stick figure in a box, but we are open for better suggestions.

Let’s add an ‘order shipped’ email feature to our book web site:

> **Send order confirmation emails**
> The system actor checks the order database for newly shipped orders.
> The book database returns a list of newly shipped orders.
> For each order, the system actor requests the related user from the user database.
> The system actor sends a confirmation email to the user, informing the user his order has just been shipped.

And an accompanying use case diagram:



4.9.2 Agents as actors

agent as an actor

Notice that the system is making the decision to initiate this functionality on its own, or autonomously. That sounds a lot like what agents do. So what happens if we think of the system as consisting of a set of agents, instead of just ‘the system’? Well, two things really. First, we can make each agent a system actor, which allows us to group related system use cases, just like normal actors group normal use cases. Second, we can make the agent the target of a normal use case. This gives a first, high-level grouping of the functionality of the system. This might not be that important for small systems, but we think it is very helpful in larger, often distributed applications. It allows us to make a piece of the application ‘responsible’ for a block of functionality, much like we assign a responsibility to a person when describing business processes.

Let’s assume an order agents sends out order confirmations once books have shipped. In addition, we’ll have functionality to view information on certain book, and have that include information about top selling books. An agent based use case diagram for this kind of system could look like this:

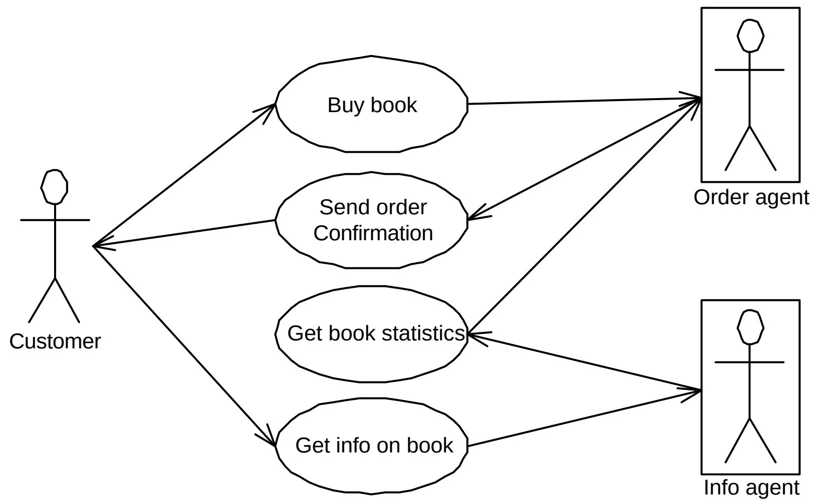


Figure 18: Book application, agent based.

This picture looks a lot more dynamic than the previous one. The order agent is actively sending out order confirmations, and the info agent is even initiating a use case, 'get book statistics', on another agent. Modelling a system like this allows us to much easier capture the dynamics of modern, large, often distributed systems. It even allows us to easily capture the functionality of large parts of a system that do not have any user interaction all together. The key advantage is we get a simple, straightforward way of showing asynchronous and autonomous components in the system. This is very important, because it much clearer communicates the dynamics and complexity of the system, and at an earlier stage in the design.

Unlike an actor, an agent is part of the system under examination. In fact, this simple equation shows it all:

actor + object = agent

agent stereotype

That is, an agent is both an actor and an object[7]. We already argued in 4.9 that an agent is an actor, and, since it is part of the system being modelled, it should be an object as well. To reflect this duality for an agent class, we introduce a new stereotype, which is, predictably, 'agent'.

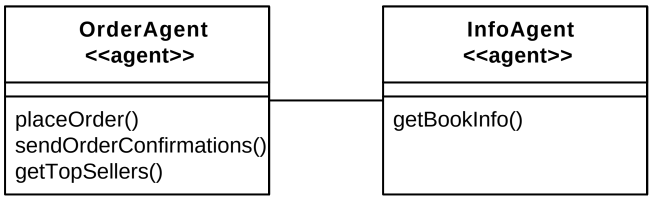


Figure 19: Agents using 'agent' stereotype

[7] Our 'equation' is just for illustrative purposes. It is not an equation in the strict mathematical sense.

component

Only one object? Well, yes, an agent is a single, identifiable part of the system, so it should be represented by one object. However, by their very nature, agents are complex objects, so they might in fact be an aggregate of several different objects. Another way to put this is to say an agent is an (active) *component*. A component is a reusable set of classes that belong together. For instance, Java beans and ActiveX objects are examples of components. If agents use a standardised language such as KQML to communicate, they even have the connect-and-use characteristic of component-based software [36].

4.9.3 Different system types

When does it make sense to model using agents? To make this clear, we recognise three different kinds of systems:

- **Reactive systems**, which do something only when someone or something interacts with it.
- **Proactive systems**, which also employ some activities on their own.
- **Agent based systems**, which consist of several different components that largely operate autonomously.

Simple single-user applications (systems) are often of the first category, and UML is very suitable to model those kinds of applications.

However, as applications get larger and smarter, there is a need to recognise *autonomous* components inside a system. A proactive system has a few of these components, which we call agents, while an agent based system is designed as consisting entirely out of such components.

UML is by itself not suited to model such applications, mainly because it only deals with 'just' objects. Therefore, we introduce an enhancement to object orientation and UML: animate objects.

4.10 Animate objects

To show how agents affect the other parts of UML, we need to introduce an additional concept, animate objects. Agents are an example of animate objects, and this addition enables us to much clearer show the dynamics of an agent based system. Other authors have also noted this omission from object orientation, in particular in relation to agents. See for instance [36].

4.10.1 Introducing animate objects

animate object

In the real world, you can have *inanimate* ('dead') objects such as books, rocks, couches and garbage cans, and *animate* ('living') objects like cows, clocks, plants, computers, humans. The big difference being an animate object can do stuff on its own, and an inanimate object cannot.

This ‘stuff’ can be as simple as to grow or to tick, or as complex as having a conversation with another animate object. Note that to avoid ‘religious’ discussions whether something is alive or not we choose to use animate or inanimate. Most people do not consider an alarm clock to be alive, but it can still do stuff on its own: It will wake you at a certain time. It is this distinction that is important.

We think it is very often useful to make this distinction when modelling as well. An animate object is thus an object for which it is important that it does something on its own, it has a degree of autonomy. Objects were characterised by state and behaviour. An animate object adds autonomy to that. We will call such an object an animate object, to distinguish it from a normal (implicitly inanimate) object. The name ‘active object’ or ‘concurrent object’ is sometimes used in other literature.

Although object orientation by itself does not preclude animate objects, objects tend to be inanimate by default. UML has actors as the key animate elements, which interact with a set of generally inanimate objects that together make up a system.

Consider you enter restaurant that works like this:

> You walk into the restaurant take a seat. You call a waiter, and give him or her your order. The waiter goes to the kitchen and tells the cook your order. The cook goes out to various shops to get the ingredients for your meal, goes back into the kitchen, prepares the ingredients and cooks the meal. The waiter, still waiting[8], picks up the plate and brings it back to you.

Sounds a bit odd? Well, this is how most (reactive) object oriented systems work. However, it is not how a typical restaurant works. In a restaurant, both the cook and the waiter are more proactive. The cook would probably makes sure he orders in advance the necessary ingredients for all dishes on the menu, and the waiter might see the table you are sitting on is no longer empty, and call at your table without you asking. Also, the waiter probably waits other tables while the cook is preparing your meal and that of other people. Again, there are two different types of objects in this interaction. You, the waiter and the cook are all animate objects, while the rest of the stuff is inanimate.

Our distinction allows us to much easier model interactions like this, resulting in more pro-active and dynamic systems. It makes modelling using objects somewhat similar to creating business processes. Those processes also tend to have a lot more dynamics and parallelism than the simple sequence diagrams we’ve shown so far. People and machines might do things in parallel, and initiate actions that are not directly motivated by events happening. Since object orientation is supposed to model reality, it is almost surprising that it is not very suitable to model interactions like this.

[8] Well, they’re not called waiters for nothing ☺

4.10.2 Class diagrams and animate objects

animate stereotype

We argued above that the distinction between animate and inanimate objects is very important, and hence we want to be able to show an object is animate in our model. To do this, we introduce our own stereotype: ‘animate’. Stereotyping a class as ‘animate’ thus indicates that instances of it are animate objects. Figure 20 shows a simple class diagram using this stereotype.

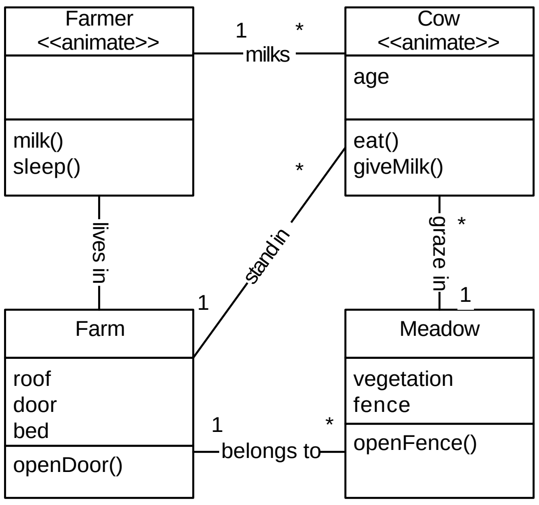


Figure 20: Simple farm class diagram.

This diagram shows two animate objects, ‘Cow’ and ‘Farmer’, and two inanimate objects, ‘Farm’ and ‘Meadow’. Notice that the methods on both ‘Farmer’ and ‘Cow’ are activities they can undertake on their own. We expect this to be quite common. In the future, the need might arise to distinguish the activities of an animate class from ordinary methods, but for now we’ll leave it like this.

UML does include some standard stereotypes that actually do mean the object in question is an animate object, for instance, ‘process’ and ‘thread’, and the standard business extension includes stereotypes like ‘actor’ and ‘worker’. In respect to those, the ‘animate’ stereotype can be seen as the base class[9] of all those stereotypes.

4.10.3 Sequence diagrams and animate objects

The real difference between animate and inanimate objects can be seen in sequence diagrams. We illustrate this using as simple example: doing the laundry.

[9] Note that we now use a modelling concept to describe a construct in the modelling language itself. It can be come even trickier: UML also defines a stereotype called ‘stereotype’, which can be used to stereotype classes that serve as stereotypes for other classes. Are you still with us?

In the old days, laundering was done by means of a tub of water, a washing board, a brush, and soap. These objects are all inanimate. An animate object, in this case a human, is needed to get any washing done.

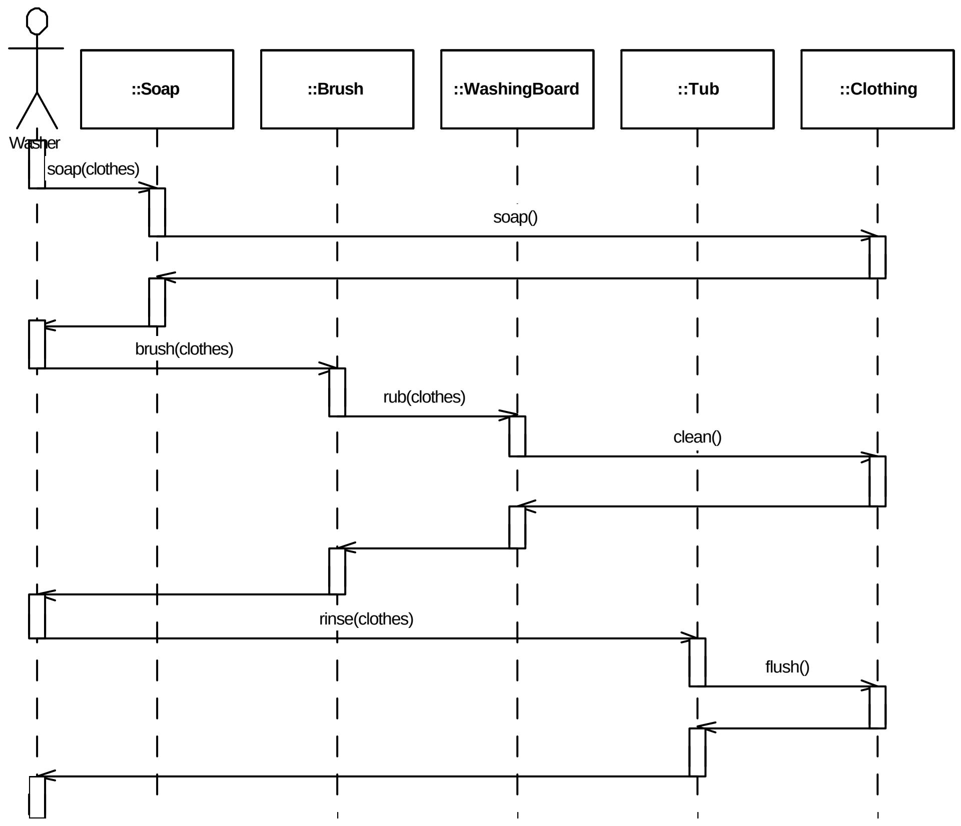


Figure 21: Old-fashioned washing.

Nowadays, we put the clothes in the washing machine, add washing powder, turn some dials, and push a button. The machine, which is an animate object, happily starts washing while you walk away. You still need to do something, but your presence is not required all the time.

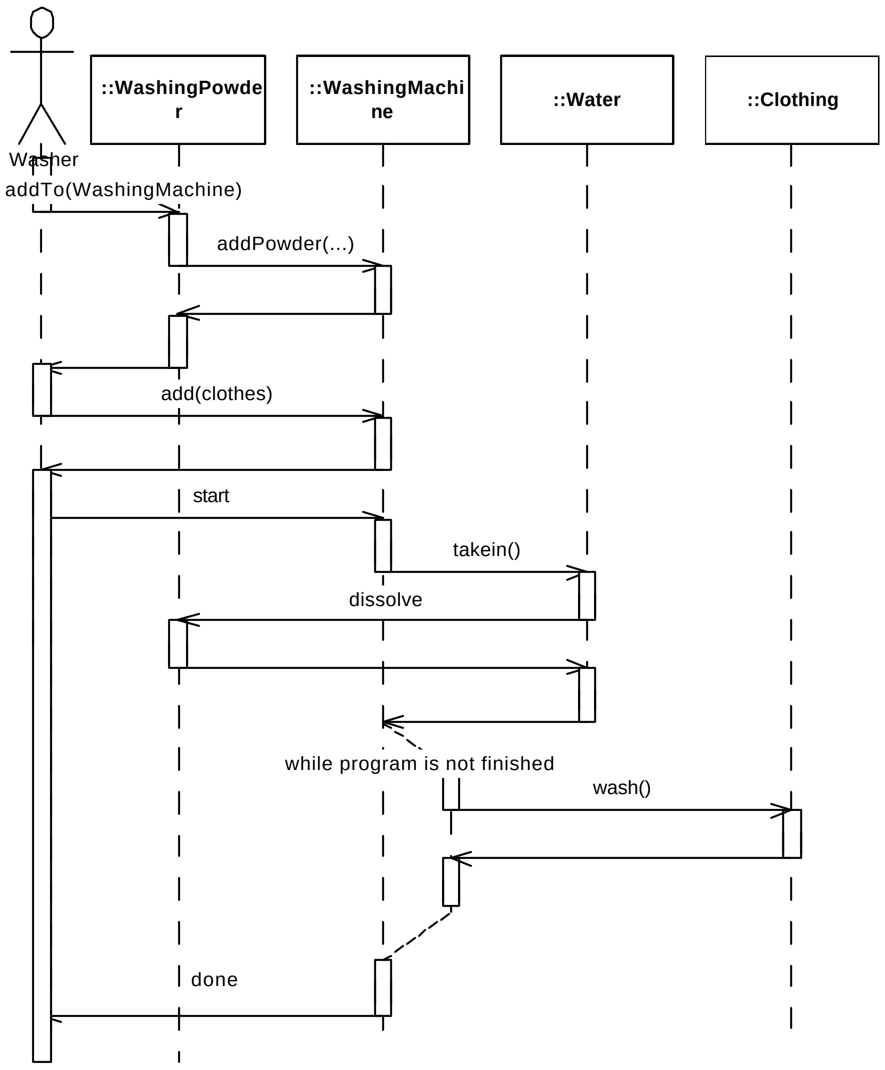


Figure 22: Washing using a washing machine.

So far, so good. However, what if you create a model for the first example like this:

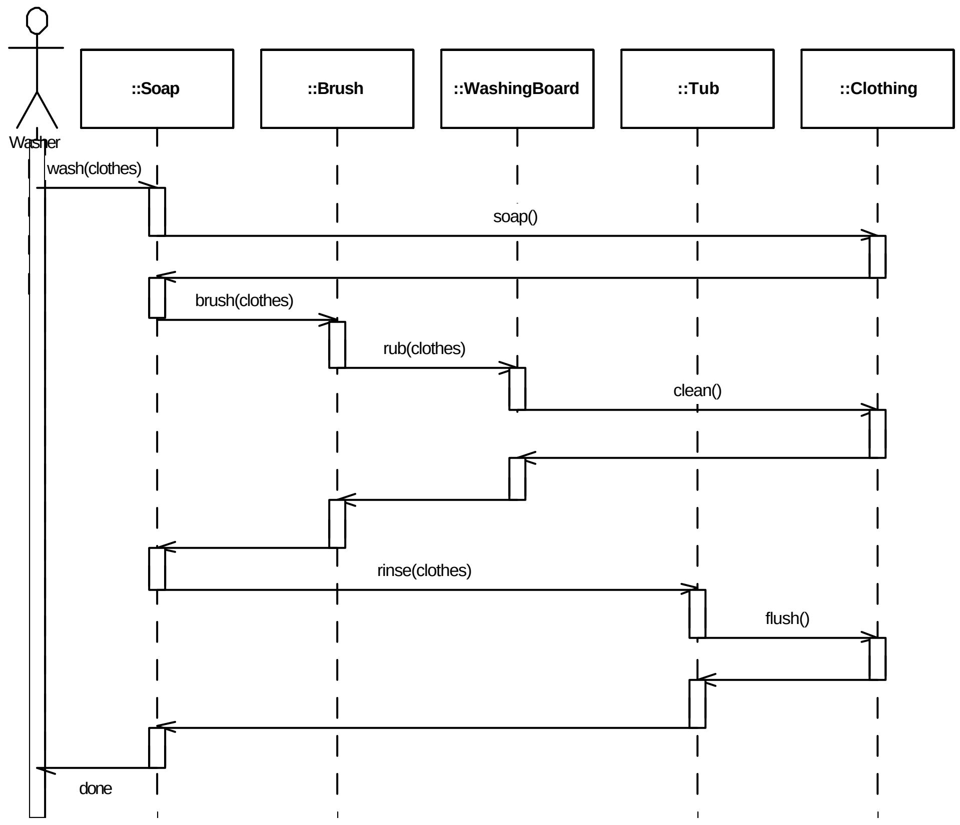


Figure 23: Let the soap do the work.

Although nothing prevented you from making this diagram, it just doesn't work. The soap just doesn't have any way of responding to your asynchronous message. It would by like putting a note on the soap saying 'please wash' and hope someone passes by at some point that actually does the washing. Now, in this example, you might be able to tell it wouldn't work like this, and model it differently.

magic activation

The trouble is that for arbitrary classes, nothing in UML helps you to determine which messages works and what will not. For the washing machine in Figure 22, and the soap in Figure 23, there is a 'magic' activation of the machine in response to the asynchronous message. Nothing in the model tells you that it is capable of responding to this kind of message, and why this works for the washing machine, but not for the soap.

The fundamental difference between animate and inanimate objects is their activation model. As said, animate objects are autonomous, the can do things on their own, they are always active. In contrast, Inanimate objects live 'on borrowed time'. That is, they need another animate object to activate them.

Using the distinction between animate an inanimate objects, we can give some simple rules as to how the different message types can be used:

- An object has to be activated to be able to send a message.
- Any object can *send* any message type.
- Animate objects can *respond* to any message type.
- Inanimate objects can answer simple messages, but can only respond to synchronous or asynchronous messages when they are activated by an animate object.

two lifelines

We will model the difference by giving an animate object *two* lifelines. The first (left) one is the normal lifeline any object has, the second is the object's autonomy line, the object's own activation. This second line will generally start activated. Using this change, let's see how the different message types work out between the different object types. Note that the whole point of a message is that something is done by the other object. An answer from the other object might not always be required, but we'll show the object answering, to illustrate it really gets a chance to do something.

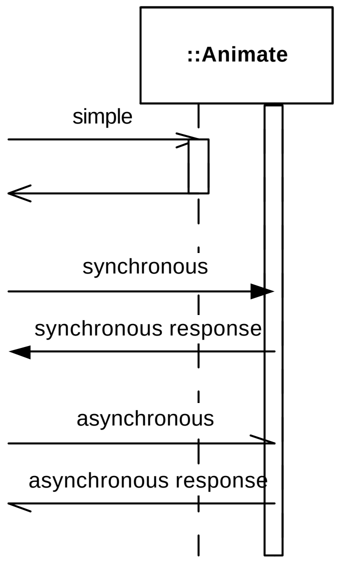


Figure 24: Animate object responding to the different message types.

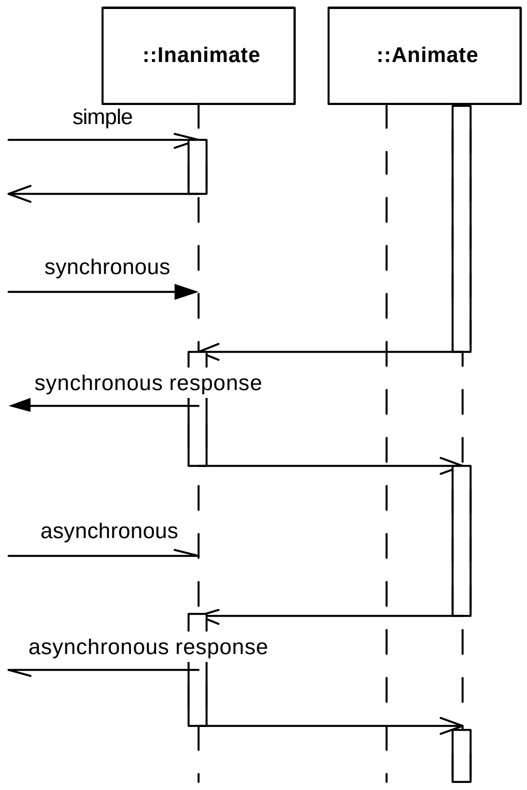


Figure 25: Inanimate object responding to messages, with the help of an animate object.

These two diagrams actually shows why ‘simple’ is such a good name for the first message type. Because there are no animate objects are involved, it looks a lot simpler.

Now, let’s modify the washing machine example. The washing machine obviously is an animate object, it does the actual washing on its own:

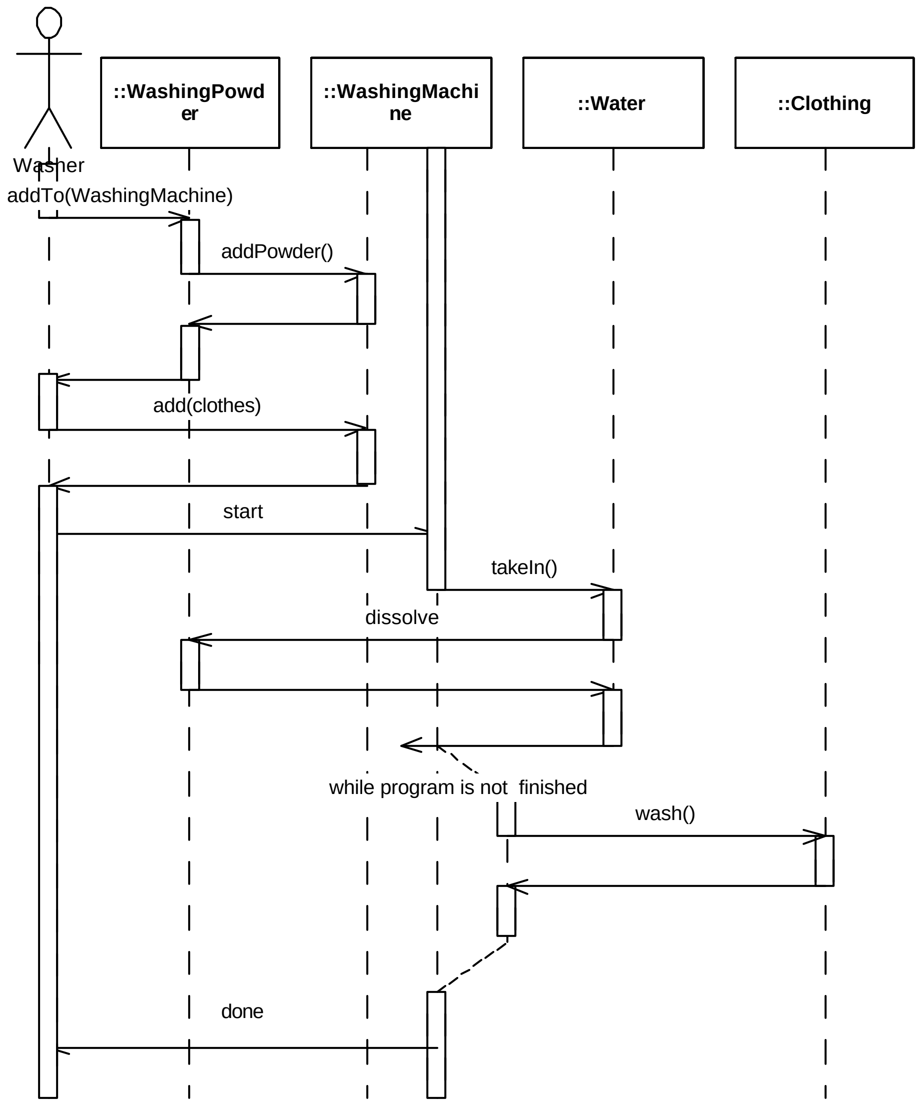


Figure 26: Washing machine as an animate object.

This picture should make it a little more clear how interaction with an animate object works. Of course, a modeller still has to know or decide which objects are animate and which are not. You can make the clothes animate objects and have them respond to asynchronous messages. That still doesn't make any sense. But at least with this convention, there aren't any 'magic' activations.

It seems a system with only inanimate objects can never be active. That's true to some degree. A system with only inanimate objects can only be activated by an actor, which can be seen as an animate object outside of the system modelled. The actor can pass its activation to an inanimate object, and so start an interaction. Just look at the old fashioned washing example again. A system such as that is purely reactive. It responds to actions of the actor, but it does not answer back on its own. You can see in the diagrams above that we need an animate object to send asynchronous messages back, which can very well end up being sent to the actor. Such a system is a proactive system.

4.10.4 Objects in different environments

There are occasions when one object (or an actor) is unable to transfer activation to another object, even if it wanted to. There might be a physical limitation, or there might be 'rules' against it. Example: ordering a drink in a bar. Even though you could physically handle the equipment behind the bar, doing so would probably mean you are thrown out of the bar. Let's make a sequence diagram out of this:

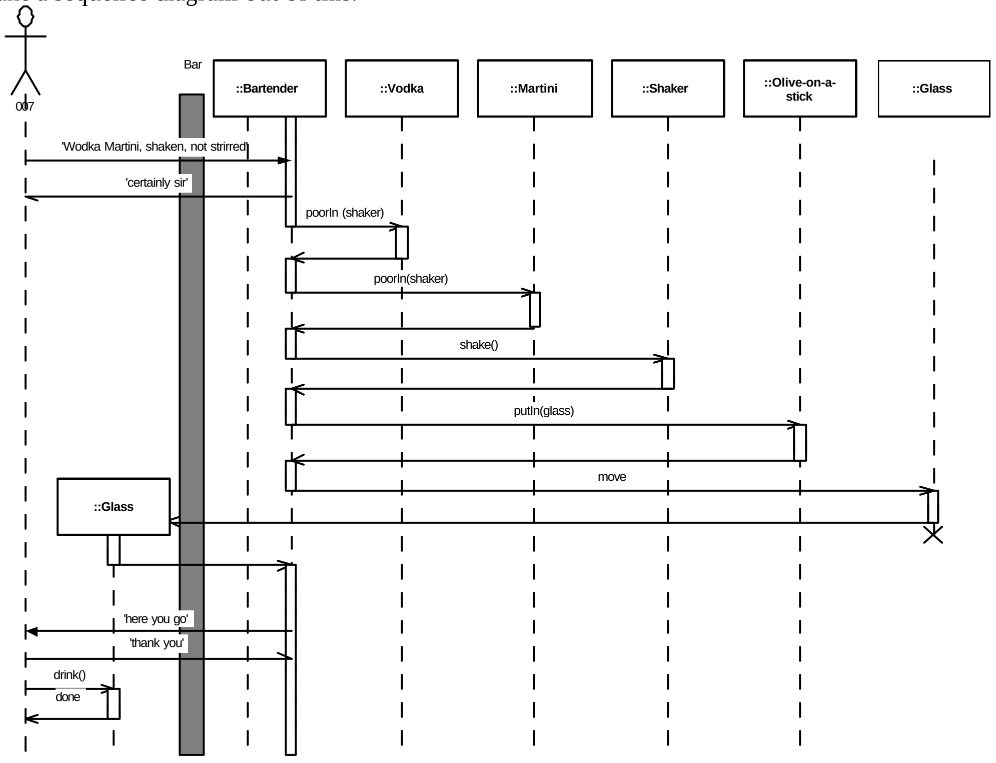


Figure 27: Ordering a drink.

The dashed bar in the diagram literally represents the 'bar' in this example, the border over which the actor[10] cannot transfer activation. The bartender, an animate object, responds to the messages of the actor and prepares the actual drink. Another curious aspect is the destruction and creation of the 'Glass' object. We use this construct to show the object moves over the bar, after which it can be manipulated by the actor.

[10] Which in this case is an agent, though not the type we've been talking about most of the time.

environment border

If, as in this case, activation cannot be transferred, we want to say the objects are in two different *environments*. In sequence diagrams, we will show this by drawing a dashed bar, the *environment border*, between the two objects. We'll call objects in different environments *separated*.

separated objects

If two objects are separated, then the only communication possible is synchronous or asynchronous messages. This then again implies that each environment needs to have at least one animate object, one that responds to those messages or activates inanimate objects for which the messages are intended, allowing those to respond.

4.10.5 Animate objects in software

As illustrated in this chapter, UML is mainly used to model software applications. A typical interaction diagram looks something like this diagram depicting an interaction with a simple drawing application:

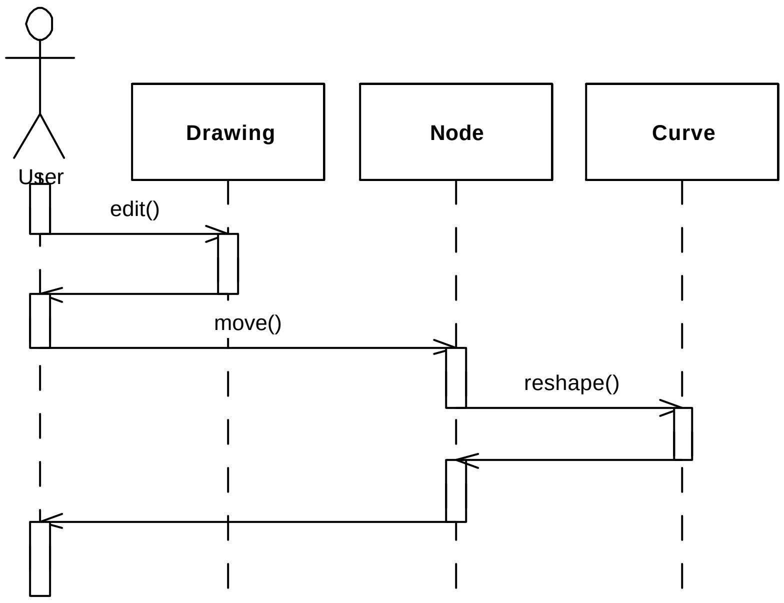


Figure 28: Simple diagram of a drawing application.

The objects in a UML model become objects living in the computer's memory. Those objects cannot be directly manipulated by the (human) actor using the application, at least not in the same way a human can manipulate objects around them. To interact with the objects inside the computer, a human uses the keyboard and other input devices to send (electric) messages into the computer, and the computer sends video and audio signals to the user. Again, we are dealing with objects in two different environments. From the discussion above, this means there should be an animate object in between somewhere, and there is. For a GUI based application, there is an event-dispatcher, which receives events from various locations and sends them to the appropriate window. Somewhat like this:

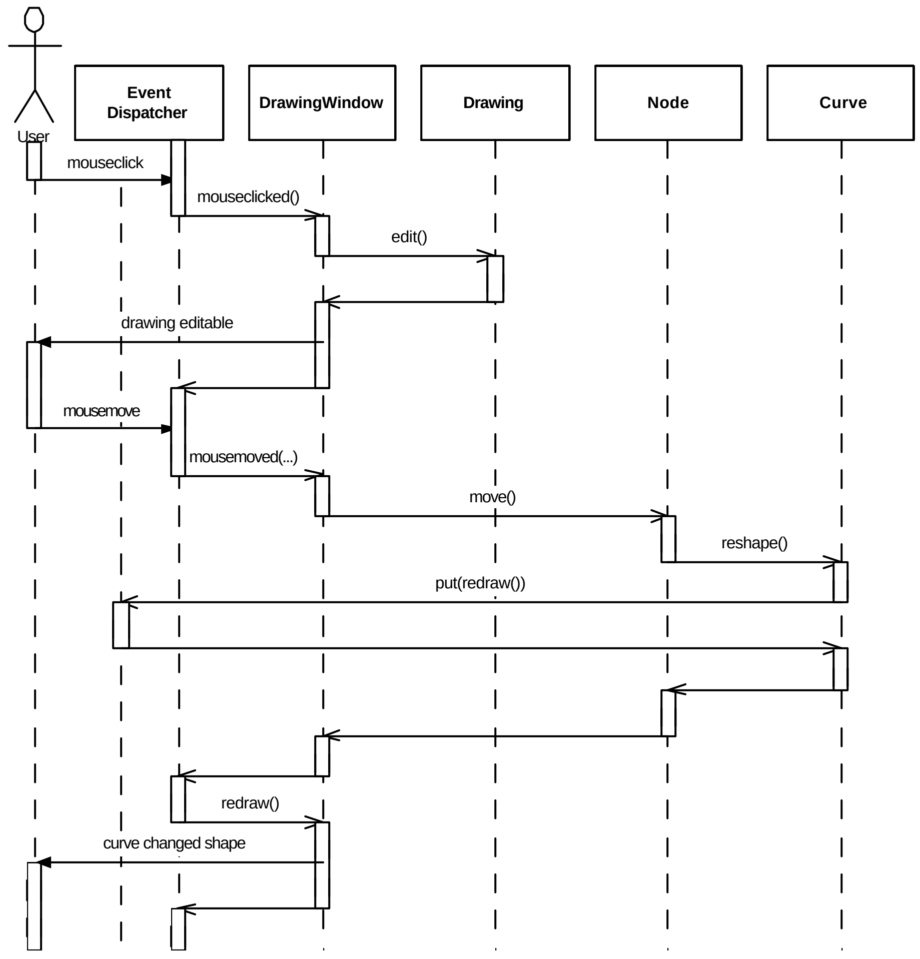


Figure 29: The same interaction, now showing an event dispatcher and synchronous messages.

This interaction shows the explicit involvement of an event dispatcher and a drawing window. We see that the user actually sends synchronous messages, and that the drawing window returns those messages (by drawing on the screen). In the first instance, the window automatically redraws itself. On the second, the curve knows the interaction changed its shape, so it issues a 'redraw' to the event dispatcher. While the drawing window would probably be part of a real sequence diagram (in a later stage of the first diagram) the event dispatcher would not. It is not part of the design of the application, and would just unnecessary clutter the diagram, so they are abstracted away.

Frequently, objects like this event dispatcher work in such a way that it seems the actor *can* directly manipulate the other objects in the application, which is why it is suitable to draw the interaction as shown in Figure 28. Other technologies, such as distributed object technology or message oriented middleware also employ these kinds of 'dispatcher' objects. We wanted to show them to illustrate that there are already many types of animate objects in software today.

So what makes these objects animate? Simply put, an animate object in software is an object and a thread, so:

thread + object = animate object

thread

A *thread*, in UML terms, is a single string of activations through a system. An application always starts with one thread, but it might spawn many more during its running. (For a more detailed explanation of threads, see 6.3.1.)

The animate objects we mentioned, like the window event dispatcher, generally have their own thread, and the thread is running in a tight loop while it is not servicing any use case. For the web server, a user's request will actually be handled by a separate thread (animate object), which allows the web server to start servicing another user's request.

In sequence diagrams, the second lifeline in the diagram literally represents the object's own thread. The sequence diagrams are very suitable to model the interaction between different threads. They allow you to show how the interaction is supposed to go, and also illustrate which objects are manipulated by different threads, and thus need very careful implementation to make sure the interaction indeed goes as designed.

We have tried to get the animate objects out of obscurity because we believe that employing more animate objects in software could make a difference. Multiple threads are currently added to a program almost as an afterthought, or at least not explicitly added to the UML model. It is often thought that building multithreaded applications is difficult. This is true, also because the animate objects we mentioned so far are fairly complicated objects. However, this is also because their activation runs along many different objects. Adding an animate object that does something on its own makes designing that interaction relatively easier, at least easier than integrating the same functionality into for instance the user's interactions.

A typical example is spell checking in a word processor. Putting that functionality in the sequences when the user enters characters is non-trivial, each time the user enters a character, the application has to check if a word has ended, and if yes, see if it is spelled correctly. It can't take to long, because otherwise the user doesn't see the results of his keystrokes. But adding a spell checker (animate object) to the application is not that hard, the spell checker can just sit in a loop and each time check if a new word is added to the document, and if yes, check the spelling of that word (or the entire sentence). It can take as much time as it pleases (within boundaries) because the user generally does not need immediate results. In essence, one complicated sequence diagram is split into two simpler ones. The price is of course that the spell checker and the GUI have to carefully *synchronise* their access to the document, as we do not want the spell checker to flag a word as spelled wrong when the user has just corrected it. Modern word processors generally implement their spell checkers this way.

Technically, a thread does not have to be part of an object, but we have found it advantageous to group them together and model using animate objects. It gives the thread a ‘home’, and gives a very good way of display of figuring out all the concurrency issues. Also, giving each animate object its own thread isn’t the only way to get animate objects, there are a few other ways, which we show in 6.3.

4.10.6 Agents as animate objects

One of the main reasons we discussed animate objects so extensively is because agents obviously are animate objects as well: they do things on their own. When is something an agent and when ‘just’ an animate object? Well, this is not so easy to say, given the wide applicability of the term agent. If we do some math, we end up with:[11]

agent – animate object ⇔
actor + object – thread - object ⇔
actor – thread.

So the difference between an agent and an animate object is equal to the difference between an actor and a thread.

What is an actor more than a thread? It associated with a set of use cases. We think if an object initiates use cases, it should definitely be called an agent. If not, do so if it seems helpful. For instance, 2.5.1 mentions a type of agent that is a character in a system. It even gives the alternative name ‘actor’ for that type of agent. So for such a system, it definitely makes sense to label an object as an agent.

As soon as it makes sense to define use cases for an animate object, the fact that it is an animate object is no longer an implementation detail. The object provides then important functionality, it has a true responsibility within the system. This is especially the case if the use case is has effect outside of the system, that is, the agent sends messages to actors outside of the system. Beware that this is a guideline, not a rule. Whether something is written and included as a use case is ultimately a choice of the modeller, as is the choice whether to call something an agent or not.

We said an agent consists of several different elements in 2.6. How are those components reflected when we view the agent as an object? The heart makes the agent autonomous, thus an animate object. The actions an agent takes are reflected by the different use cases initiated by that agent. To be able to take those actions, it needs sensors and effectors, which should be reflected by methods on the object. The memory of the agent is, fairly straightforward, represented by different attributes on the agent. Finally, the brain. The brain itself cannot be reflected on an object diagram, However, a state diagram (4.6) is a very useful tool to show when an agent does what.

[11] Again, this might not hold in the mathematical sense. We belief this outcome to be true nonetheless.

4.10.7 Agents and state

Animate objects and especially agents often have fairly complicated behaviour, which often changes depending on their internal state. Therefore, both are good candidates for getting a state diagram. Incoming inputs might change the state the agent is in, causing it to take actions. Even when the state doesn't change, but the agent still takes action, this can be shown by a transition into the same state (a degenerate state change), with in its description the event and the action.

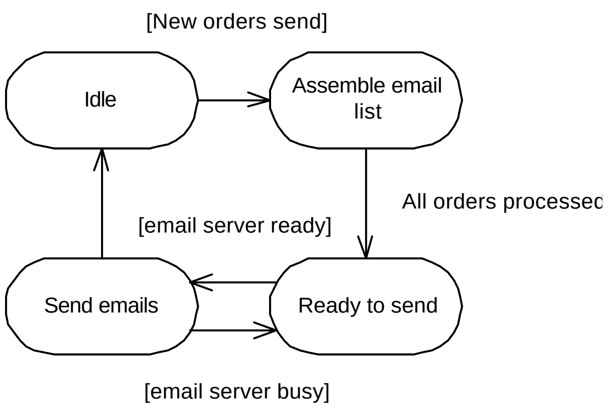


Figure 30: State diagram for order agent sending emails.

This is a state diagram for the order agent in 4.9.2. The agent first collects all new orders, then assembles a list of emails grouped by person. Once it has processed all the orders, it is ready to send out all the emails. It then tries to send all the emails. However, it might be that the email server is busy, (or unavailable), so it temporarily becomes 'Ready to send' again. Once it has send out all the emails, the agent becomes 'Idle' again.

Together, the state diagrams and sequence diagrams can describe how an agent operates. The state diagrams show what happens when, and the sequence diagrams detail what happens with each event.

4.11 Conclusion

In this chapter, we have shown how you can model with agents in an object-oriented way. We have briefly skimmed over UML, the Universal Modelling Language, and introduced a few additions to UML to better enable us to model agent specific details.

UML consists of several types of diagrams, and the UML methodology describes what to use each diagram for. UML is object oriented, meaning it is based on the idea that everything can be seen as an object and thus modelled as such.

- **Use cases** are used to describe all the functionality of the system. Each use case describes a possible interaction with the system. Each use case is initiated by an actor. Different actors represent different types (both human and external systems) of the system modelled.

- **Class diagrams** show the different classes (objects) that together make up the system. Each object has zero or more attributes, information about the object, and methods, what the object can do.
- **Sequence diagrams** show how objects interact with each other. Each use case should have a corresponding sequence diagram showing how different objects in the system interact to provide the functionality described in the use case.
- A **State diagram** shows how different events affect the internal state (one or more attributes) of an object, which is useful to show if the behaviour of the object varies depending on the internal state.

These diagrams could be created in this roughly this order, although the UML methodology recommends an iterative approach. There are a few other diagrams that we haven't shown because we have no need for them in this thesis.

We recognise three different kinds of systems:

- Reactive systems.
- Pro-active systems.
- Agent based systems.

The first only responds to outside stimuli, the second has some autonomous components, while the third's largely operates on its own, asynchronous to outside events.

UML is by itself not very suited to model autonomous components, which is why we introduced a few enhancements to UML. We (and others) say that while it is true that everything can be seen as an object, it perhaps makes sense to distinguish between two different types of objects, animate and inanimate objects. It is fairly easy to see this distinction in real life as well: inanimate objects don't do anything by themselves, while animate objects do things on their own. While UML or object orientation doesn't preclude the fact that objects do things on their own, the lack of modelling capability means objects tend to be inanimate by default.

We introduced several enhancements to UML to allow us to model using agents and animate objects:

- An agent is both an object and an actor, which means it is able to appear in use cases and initiate other use cases. As a guideline, animate objects that initiate use cases are actors, they initiate functionality of the system.
- Two stereotypes, 'animate' and 'agent' to distinguish those kind of objects from other types of objects in class diagrams.
- Animate objects in sequence diagrams get an extra lifeline, to indicate they can do things on their own rather than just being activated

(indirectly) by an actor. This also allows them to respond to synchronous and asynchronous messages directly.

- When objects reside in two different environments, we call them *separated.* Two objects that are separated cannot transfer activation to one another, only exchange synchronous or asynchronous messages. We show the separation by introducing an environment border, dashed bar, in a sequence diagrams.

5

CHAPTER 5 Mobile agent paradigm

We already mentioned mobile agents in chapter 2, and briefly said what they are and what makes them so interesting. In this chapter we explore these facts deeper, and explore what it means to use mobile agents as the foundation for a distributed application. We consider mobile agents to be a new and promising paradigm for building distributed applications. In order to stake this claim, we show how mobile agents compare to other distributed computing paradigms and how mobile agents create increased flexibility and network awareness for distributed applications. After that we illustrate all this with some sample mobile agent architectures and discuss some interesting applications for which mobile agents can provide an edge.

paradigm

Note that when we say paradigm, we mean it in a light sense, as ‘a prototypical idea’, and not in the heavy sense associated with ‘paradigm shift’. A new paradigm like this might *cause* a paradigm shift if it is widely accepted as, in this case, a new and better way of developing distributed applications, but it doesn’t have to.

5.1 Distributed Computing

Before we can discuss different distributed computing paradigms, we need to explain briefly what distributed computing is in general, and what the main issues are you face when designing a distributed application.

distributed computing

So what is *distributed computing*? In distributed computing, several computers, connected by a network, work together to accomplish some task. Those computers work together because it is more convenient or simply necessary to do so. It is convenient if the other computer is better equipped to do something (faster, more memory, less busy, etc.), and necessary when neither has all the resources needed for the task. For instance, for displaying a web page, one machine (the web server) has the page to display, but the second (the machine running the web browser) has the screen to display the page to the user.

Today’s world would be vastly different without distributed computing. Browsing the web, reading and writing email, and multi-player network games all utilise some from of distributed computing, and most companies today would be helpless without their computer network and the applications that run on it.

client-server

The web browser example shows the most well known form of distributed computing, that of *client-server*. A client-server system is partitioned in clients, the pieces that drive the interaction, and servers, pieces that provide particular services and passively wait for a client to interact with them. Those clients are generally users behind a client application on their machines. Distributed computing is more general, and does not assume any machine plays a particular role. Still, it is often useful to call a machine initiating some interaction the client, and the one responding the server. These roles might very well be reversed in another scenario.

5.1.1 Advantages of distributed computing

There are many good reasons to do distributed computing, and most new applications build today are capable of some form of distributed computing, if it were only to connect to the company home page from the help screen. We list a few of the most important advantages:

Physical distribution
The biggest advantage of distributed computing is without a doubt the physical distribution of users and connected devices. It enables the creation of ‘cyberspace’ and the resulting ‘death of distance’. That is, a well built distributed application makes it just as easy to communicate to someone or something on the other side of the planet as if that someone or something was right beside you.

Load distribution / parallel processing
Another advantage is the distribution of processing among more than one machine. With high-performance machines, it is often a lot cheaper to buy two machines with power X than one with 2X. So it makes economic sense to buy two (or many more) machines and invest a little to make them do the processing together. Today’s processing needs are so substantial it is impossible to do them on a single computer.

Take the new Star Wars movie ‘The Phantom Menace’. A single frame of digital effects probably takes a few hours to render on a pretty powerful Silicon Graphics machine. 95% of the film is digitally altered in some way, and that’s an awful lot of frames. To render all those frames, Industrial Light and Magic (ILM) uses a rendering farm, a huge cluster of equal machines that can render many frames in parallel. Still, The Phantom Menace used half (ILM does do other things) of that capacity for a whole year.

Reliability
More machines can improve reliability and availability by having several machines that can take each other’s place. The chance of them failing all together is a lot less than that of the chance of failure for a single machine. Also, you can easily take one offline for maintenance or upgrade and still keep the application running. This is for instance crucial for today’s e-

commerce applications, which need to be up 24x7, since the Internet never sleeps.

Specialisation
Using more than one machine means each machine can be best equipped for its task. A database server above all needs big fast hard disks, a number crunching machine needs an array of super fast processors, and a graphical modelling workstation needs specialised 3D rendering hardware.

5.1.2 Issues with distributed computing

Distributed computing of course comes at a peril. Some things you take for granted when designing a normal, stand-alone application suddenly disappear when you do the same thing distributed. The most apparent thing is that two pieces of a distributed application on different machines inherently run in two different processes. (A process is a running instance of a program on a computer. For more explanation, see 6.3.1.)

No shared state
First and foremost, two processes are totally oblivious of one another unless they are enabled to communicate in some way. They could do so by for instance reading and writing shared files on a disk, or by operating system support for shared memory or named pipes. For two pieces of a distributed application it is the same, but now the communication also involves network communication.

Inherent concurrency
Second, two processes run independently, so there is inherent concurrency, which makes it more difficult to co-ordinate and synchronise. However, this concurrency, and the underlying autonomy is an advantage as well, as it allows for parallel processing.

Semantics
Suppose the pieces of some operation are spread across several machines. If one of those machines fails, the other machines, until the machine is available again, have no way of finding out how far the crashed machine got with its pieces of the operation. This means that the *semantics* of such an operation are different depending if it is executed on one machine or several. This is especially apparent with transactions, compound operations that have to succeed or fail as a whole. Managing distributed transactions is much more difficult than single machine transactions, and requires the implementation of a complicated transaction protocol.

Platform differences
Finally one thing that is unique for distributed applications is differences in terms of hardware or software, which means any communication between two components on different systems has to be very well defined and possibly translated to a format understandable to the rest of the system.

5.1.3 Network communication

In the absence of a shared state, distributed applications rely on computer networks to communicate with each other. On a physical level, there are many options, like Ethernet, the telephone network, ADSL, fibre optics, satellite links and radio waves, to name a few. No doubt newer, faster, and further reaching methods will be devised, maybe some day allowing inter-stellar communication at the same speed we can now conduct phone calls[12]. Even though network speeds are getting faster and faster, transferring a chunk of data over a network is always an order of a magnitude slower, and sometimes much more expensive, than accessing that data in the memory of the computer. This means that the designer of a distributed application has to pay careful attention how design decisions affect the network use of the application.

5.1.4 Distributing a computation

Distributed computing means you distribute a computation. That sounds rather trivial, but there is a little more to it. It means the designer has to decide which pieces of a computation run on which computer, and as a result of that, what and how much data needs to be transferred between those computers. A poor design might result in prohibitively slow, network hogging application, even though everything looks nice and clean on paper.

There is very little code that really has to reside on one machine or the other. Only code that directly influences hardware needs to run on a particular machine to execute, like code that read or writes to files, or displays windows to the user. And even for these this is not always true. Most modern operating systems allow an application to access a file on a different machine with just about the same ease as a file on the local disk, and there are even some (most notably X-windows) that make it seem to the application it is displaying windows locally, while in reality that window is shown on a monitor connected to another machine. This means the designer in theory has a lot of freedom where to run some code. The trick is how to use this freedom well.

Where to execute some code is a typical trade off situation between processing and data transmission. Consider three pieces of a distributed computation: A, B and C, and B needs the result of A, C needs the result of B. Let’s say two computers involved in computation, and we put A on computer 1 and C on computer 2. The choice is now where to put B.

[12] Subspace communications ? Anyone ?

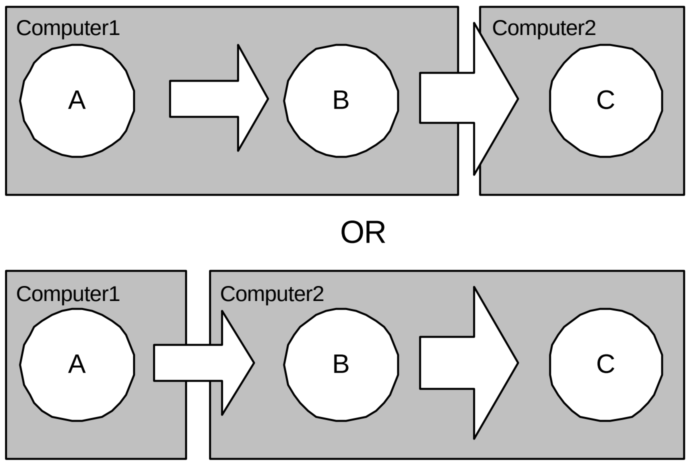


Figure 31: Distributed computation. Where should B go?

If we put B on 1, processing for A and B takes place on computer 1, and we need to move the result of B to computer 2. If we put B on computer 2, we now need to transmit the result of A to computer 2, and processing for B and C takes place on computer 2. Which of the two is better? There is no easy answer. In the picture, we've made the second arrow in the process bigger, and if take that to mean the result of B is bigger than that of A, than, with all other things equal, B on 2 would be the preferred choice, because that would minimise the network traffic between the two computers.

Unfortunately, all other things are seldom equal. B might be a very lengthy computation, which puts a big strain on computer 2 if put there. Or it might be desirable to have as little code on Computer 2 as possible. The result of B might contain sensitive information, which should not travel over a network. Or it might be hard to encode the result of A to travel over a network, but encoding B's result might be easy.

Even worse, this computation is unrealistically simple. A real computation probably looks more like the sequence diagrams shown in chapter 3, and even those are still pretty simple. If more than two computers are involved (as is often the case), differences in network speeds between different computer can make big differences. Of course hardly any system supports only one single computation, so logical (and thus nearly always physical) grouping of functionality, and re-use of sub-computations complicates things. Availability of certain machines might spice up the picture. All in all, to get it right you need to solve a hard, multi-parameter optimisation problem, and that provided you can even gather the required information beforehand, because it might be difficult to estimate data sizes and application use.

5.1.5 Security

No discussion on distributed computing is complete without mentioning security. For single computer applications, security is usually pretty simple. If you are able to use the computer, you are generally allowed to do just about anything. Sure, erasing the entire hard-drive might make the IT department extremely unhappy, since they have to re-install all the software on it. Knowing that might make you think twice before taking such drastic actions. But the point is, the computer will let you. For distributed applications, it is seldom that simple. Many different people are able to access machines running a piece of a distributed application, and what they are allowed to do differs depending on their credentials.

Computer and network security is a huge and complex topic and in requires a lot of attention when designing a distributed application. Still we refrain from diving into it further, because ultimately security is about things you *cannot* do, and we want to focus on things you *can* do. It suffices to say that all of the technologies we mention further ahead also have extensive built-in mechanisms to warrant proper security.

5.1.6 Guidelines for building distributed applications

After the previous discussion, you might be wondering how we were ever able to build all those distributed applications in the first place. Well, we of course deliberately painted a very cloudy picture. For a real life distributed application, it is often very obvious where the truly hideous amounts of information usually reside. A fairly common place is database servers, which provide generic mechanisms for searching and filtering the data in them. If those generic mechanisms are not enough, if for instance you need to find pictures in a similar shade of blue, you either have to use a particular vendor's specialised database, rely on extension mechanisms of some type of database, or hide the database server behind a specialised server process running on or near the database server that can provide the functionality.

Most design methodologies (like object oriented methodologies, see chapter 4), encourage to group and reuse functionality. Putting such a group of functionality together on one machine server means that the tight interactions and data exchange within such a group naturally occurs on one server. Another thing that often helps is various forms of caching mechanisms, so making sure that data, once fetched from a remote system, is reused by other computations instead of retrieved every time.

Of course, you do not have to build a distributed application from scratch. There is a whole mass of technologies out there that can help in the creation of a distributed application. Some examples are database servers, web servers and browsers, mail servers, network file systems, groupware, transaction monitors, and various forms of middleware.

Finally, a last thing that often helps is throwing money at it. If a server or connection is seriously overloaded as a result of some design decision, simply upgrading the server or connection, or adding one often makes the problem go away. This is actually not as bad as it sounds, since buying a more expensive machine might actually be cheaper than having the developer optimise or redesign the suffering functionality.

5.2 Modelling a distributed application

The modelling language UML and accompanying methodology we introduced in chapter 4 can of course be used to model distributed applications. The basic methodology does not change when an application is distributed. Once it is time to decide how the different components will be distributed it is probably a good idea to make different class diagrams that show which objects go on which (type of) system. If you need to show that two classes belonging to different systems communicate this can be done by placing the one class in both diagrams. Like ‘Class3’ in the following diagram:

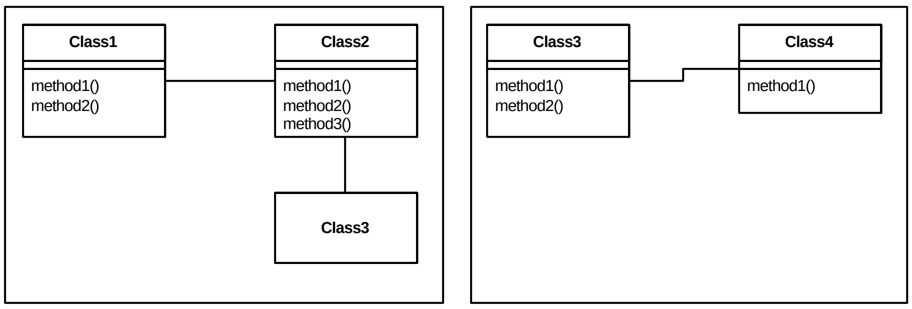


Figure 32: Two class diagrams for a distributed system, with a duplicated class.

Of course, we would not have introduced our extensions in chapter 4 if we did not intend using them in this and the following chapters.

Objects in the different machines of a distributed application live in different environments. This means, as stated in chapter 4, that they cannot transfer activation to each other, but can only send synchronous and asynchronous messages. This then again means that each machine needs to have at least one animate object, one that either itself is involved in the communication with other machines, or that periodically activates inanimate objects to do so.

process border

We will use our environment border (4.10.4) to show where messages between objects cross process borders. Because the environment border depicts the border between two processes, we will call it *process border* in this case:

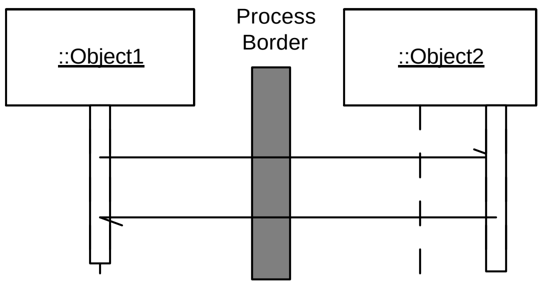


Figure 33: Sequence diagram with a process border.

We think it is important to show this difference, because, as we argued in 5.1.2, a message across a process border is a whole different ball game. By making this difference explicit in the sequence diagram, we easily distinguish the difficult messages from the easy ones, wonder if they are indeed possible as we've drawn them, or what it takes to make them work. As a general rule, we would probably want to keep the number of arrows crossing a process border to a minimum.

5.3 Paradigms for distributed computing

Here we discuss and compare several different paradigms for designing distributed applications. We will focus on what type of interaction the paradigm supports between pieces on different machines, in what way they make the different environments seem one. To this end, we use the object interaction diagrams, which help show the difference between the different paradigms. All these paradigms are high-level, in the sense that they assume that communication is between the components is reliable and meaningful, that is, a single message send to an object on another machine does not need to be decoded and / or combined with other messages.

Together with each of the paradigms we discuss some technologies that support building application based on the paradigm. Most of the technologies we mention are collectively referred to as middleware. They are called middleware because they 'sit in middle' of the different pieces of the application and make them work together. Middleware doesn't provide any end-user functionality itself, it just helps address the various issues with distributed computing in various ways, so the designer can better focus on designing application functionality.

5.3.1 Distributed objects

Distributed object technology allows you to do distributed object oriented programming. (See chapter 3 on object oriented programming) It enables you to call methods on objects that are on another machine with the same ease as calling methods on the same machine. This means interaction between objects on different machines looks like this:

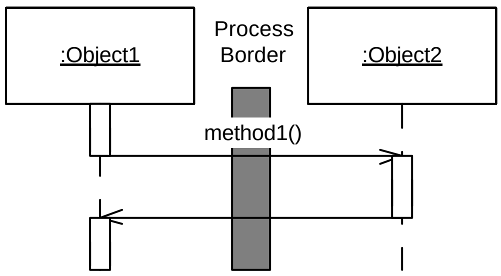


Figure 34: Distributed objects interaction.

This is of course very convenient, because it completely abstracts away the networking nightmare that goes on behind it, and makes it look like Object2 was right beside Object1. Two objects that were in different environments appear to be in one.

It is not that easy to realise a distributed objects technology, especially across different platforms. The distributed object technology has to make sure that data passed to a method on a remote object still means the same thing on the target machine. Since the machines don't share any memory, a reference to some piece of memory is meaningless on the other. Also, different machines might represent the same thing differently. For instance, a Sun Sparc puts an integer number (which is four bytes) in memory with the highest byte first. An Intel machine believes the lowest byte should go first. If you would just exchange the four byte piece of memory between the two machines, chaos arises. For these reasons, distributed object technologies require you to exactly specify the signature of each method of a remote object, and they restrict the types allowed in a remote object's methods.

We now briefly mention some of the most important distributed object technologies, hoping you don't drown in the acronyms:

DCOM
DCOM is the Distributed (the D) version of COM, which stands for Component Object Model, a programming language independent, standard mechanism for calling objects defined by other programs. DCOM extends COM by allowing the actual object to reside on a different machine. DCOM is created and maintained by Microsoft, initially only available on Microsoft platforms, although a few hazardous steps are taken to make (D)COM objects callable from other platforms. DCOM is built on top of the OSF DCE RPC protocol, whose nine capitals stand for Open Software Foundation, Distributed Computing Environment,[13] Remote Procedure Call.

The signatures in DCOM are allowed to have primitive types, structures of primitive types, pointers to the allowed types, arrays of allowed types, and pointers to interfaces of other remote objects.

[13] For those few of you that thought DCE stood for Dutch Computer Enterprise, um, yes, well, that too, but not in this case… (Dragnet rules ! ☺)

CORBA
CORBA is another nice acronym, meaning Common Object Request Broker Architecture. Unlike DCOM, it is defined by a standards committee (OMG) and from the start platform and language independent. As long as a CORBA implementation, called a CORBA-compliant ORB (Object Request Broker) exists for a particular platform, a program can call CORBA objects on any another machine. CORBA ORBs use the GIOP (General Inter-Orb Protocol) protocol to communicate, which, when implemented on top of TCP/IP takes the name IIOP (Internet Inter-Orb Protocol).

Method signatures are similar to DCOM, but CORBA does not allow pointers (because of the platform independence) and has a few more arcane primitive types.

RMI
RMI is distributed objects for the Java Platform, as created by Sun Microsystems. RMI stands for Remote Method Invocation, which is more or less RPC (see DCOM) translated into object orientation lingo. RMI is available for any platform for which a Java virtual machine is available (although Sun had to fight a little to get Microsoft to implement it in their virtual machines as well). Because of Java's platform independence, Java has some unique additional features, as described below.

Signatures are allowed to be primitive types, remote object references and normal object references. The last is fairly fancy, and means objects can be passed back and forth between machines. In the simple case, this creates the same behaviour as the passing of structures (Java doesn't have structures) in the other two examples, but it offers more advanced feature: mobile objects. RMI is able to send code along with the object's data as well, so the client machine or remote machine might receive the new code if an object of an unknown type is passed to it. This same feature also allows RMI to send code for remote objects to a client machine at run-time, which minimises the amount of code required to be present at the client machine at start-up.

5.3.2 Agents

Agents, as discussed in chapter 2, can of course be used to create distributed applications. In fact, some of the examples we gave were distributed applications. Agents, being animate objects, have a different interaction model. Instead of the synchronous model shown above, we can have asynchronous interaction, like this:

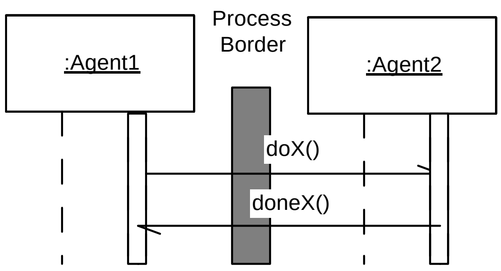


Figure 35: Agents exchanging asynchronous messages.

Now, as shown, we have two asynchronous exchanges between two agents, one initiated by each agent. This is delegation at work. Agent1 delegates the task X to Agent2, which returns its results to Agent1 when it is done.

While this interaction is slightly more complex that the previous, the agents can execute tasks in parallel, and there is no open network connection required in the mean time. This is especially an appropriate model for distributed applications, because it allows us to take advantages of the inherent concurrency of distributed systems.

The messages in the diagram shown are can use any technique, and for instance use KQML/KIF (see 2.9 for message exchange.

If we combine distributed objects and agents, we get agents that can have both activation and asynchronous messages, just like agents that live in the same environment. In particular, we can have asynchronous operations using activation, like this:

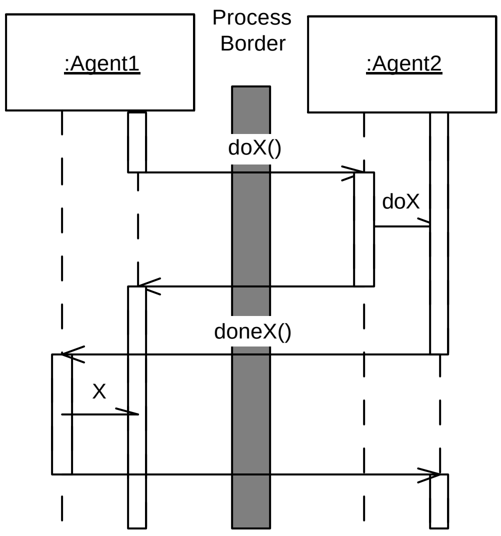


Figure 36: Agents using activation for asynchronous messaging.

This approach has two advantages: An agent can choose not to accept a message, and the other agent can take action on that immediately, rather than having to wait for a response. Second, it allows both agents to expose a nice, well-defined interface, instead of having to use a generalised messaging system.

5.3.3 Mobile code

Mobile code occurs when the data transported to the other computer is there interpreted as code. This is very nice, because it means you can have the computer do ‘new tricks’ on the fly. A very simple example is when you execute a program whose code resides on a remote file system. It gets more complicated however, if (and we see a reoccurring theme here) the machine or platform of the remote and local machine are not the same. Also, we only consider it to be really mobile code if the code is moved transparently, without the user explicitly copying and installing or compiling new code.

remote evaluation, code on demand

There are two flavours of mobile code: *remote evaluation* and *code on demand*. The difference lies in which party initiates the action. With remote evaluation, you send code to another machine to be evaluated there. This is the model used by database servers, for instance: You send your query (the code) to database server, which evaluates it and returns the results to you. The reverse is code on demand, where you request code from another machine and run it on your own. This the well known Java applet model: the web server sends the code for the applet to your machine, and the applet is run locally.

A mobile code interaction works as follows:

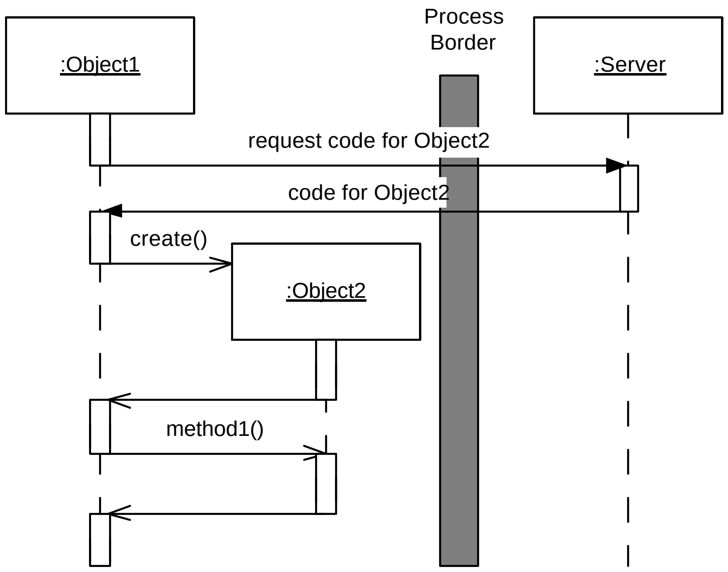


Figure 37: Mobile code, code on demand.

How hard it is to move the code depends on two factors: the portability of the code that is exchanged and the difference between the two platforms that need to exchange code. Portability means on what other platform the code can run without modification. Code portability is a sliding scale. The further up the scale, the more the language avoids platform specific issues, which means more portable. They downside is that the language is generally less powerful, and more extra software is needed to run the code. Below we explain some of the options: from less portable to very portable.

Machine code
The least portable, but most powerful, and requiring the least additional measures, is machine code. (.exe or .dll files on the Intel win32 platform). Machine code for Intel win32 of course only runs on Machines with an Intel compatible processor running win95/98/NT. Still, since this is quite a large slice of the computer pudding it is definitely an option. ActiveX controls in web pages, for instance a tree control, use mobile code to work. When loading a web page that contains an ActiveX control, the browser also loads a .dll with the code for the control.

3GL
The next option is code in a third generation programming language (3GL) such as C(++) or Pascal. This code needs to be compiled (translated into machine code) before it can be run, and compilers for said languages are available for most platforms. We have never seen this option used however, perhaps because it is generally very hard to make code that will compile and run flawlessly on many different platforms.

Java
A special case of a third generation programming language is the Java language and platform. Java sits somewhere in the middle between machine code and interpreted language, which is why it is listed here. Java code is compiled into Java byte code, a sort of platform independent machine code. Java code requires a Java virtual machine (JVM) to run, which is literally that: a Java virtual (as opposed to real) machine makes a specific real machine look like a machine that can run Java byte code. At run time, the JVM interprets the Java byte code and translates it into the machine code of that platform. This special construction allows Java code to run on any platform that has a JVM, of which there are quite a few.

The most common use of this Java code mobility is with Java applets. Java applets are small programs that are displayed with a web page, and allow for enhanced interactivity. They are similar to the earlier described ActiveX controls, but have surpassed those in popularity big time, largely because of the platform independence and because they are easier to program.

Scripted languages
A scripted language is generally a more task specific programming language, without the full power of a third generation language. Scripted languages are, like 3GLs, human readable, but unlike 3GLs, are generally interpreted, not compiled. At run time an interpreter program for a specific language examines the code and executes the equivalent in machine code.

SQL is the standard language used to query relational databases and is the living proof that mobile code is not as new as some of us might think.

JavaScript and VBScript are both scripting languages that can be embedded into HTML pages. Both allow for simple interactivity on a web page. And are highly portable. They do lack the power a full programming language like Java or C++.

5.3.4 Mobile objects

Mobile objects are sort of the flip side of distributed objects. Instead of accessing an object remotely, you make it come to you, or send it to the other side. Interaction looks like this:

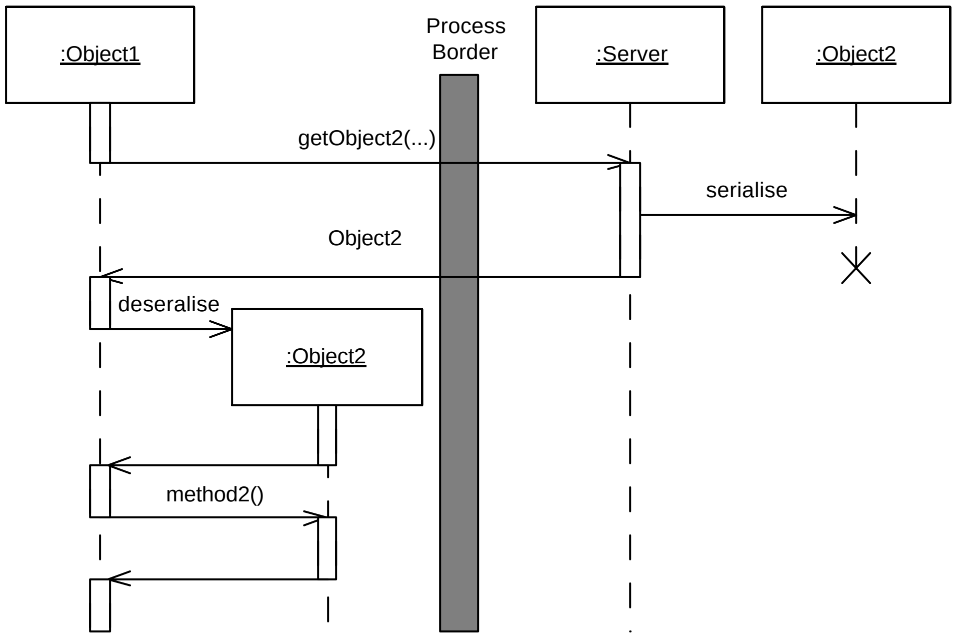


Figure 38: Mobile objects.

Now, this looks very similar to mobile code, but there is a big difference. Now, Object2 already existed before, it had a certain state, and we have to ‘import’ the state of Object2 in another process.

To support mobile objects, you need two things: object serialisation and mobile code. Object serialisation means you can write the entire state of an object to a format that is transmittable over a network and can be used to recreate the state of the object on the other side. Mobile code is needed if the receiving side does not yet have the code of the object it got.

In theory, you could have just mobile objects, but we’ve seen it only in combination with distributed objects. Combined, mobile objects allow the inclusion of (non remote) object references in distributed object signatures. This means you can execute a method on a remote object and pass it an object you have locally as a parameter. The remote object then receives (a copy of) the object, and can interact with it locally. Vice versa, a remote object can return an object from a method, and you can then interact with it locally.

They key technology currently supporting mobile objects is Java with RMI, which is thus an extremely flexible technology for building distributed applications.

5.4 The mobile agent paradigm

We've already mentioned mobile agents a couple of times, but now we really mean business. Mobile agents are a new and very promising paradigm for designing distributed applications. Mobile agents can be thought of as the merger of mobile objects and agents. As such, it allows for building extremely flexible, large-scale applications using the elegant idea of partitioning the application in several agents that can freely move around the network. Such an application consists of two basic sets of components: mobile agents and agent spaces. We haven't mentioned agents spaces before, so we explain those first.

Agent space

agent space

An *agent space* is a hosting environment for mobile agents. Every node in the distributed application has one or more agent spaces. The concept of an agent space hosting agents is similar to operating system that 'hosts' applications. In fact, an agent space doesn't have to be much more than an operating system. The key additional service it provides over an operating system is sending agents to other agents spaces and receiving an activating mobile agents from other agent spaces. This picture says it all:

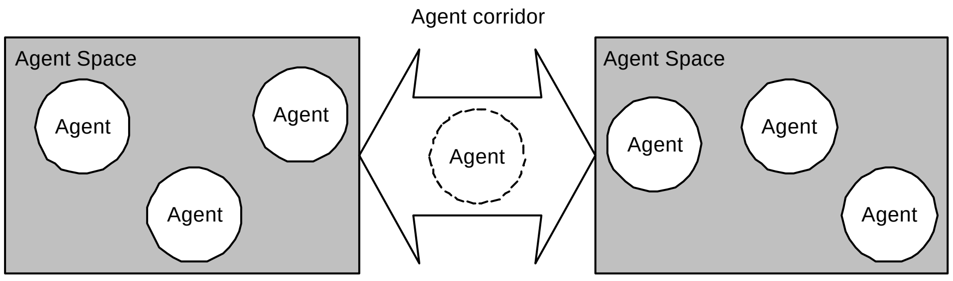


Figure 39: Mobile agents and agent spaces.

The picture shows two agents spaces, each with several agents. The spaces are able to send and receive agents to each other. This creates a sort of 'agent corridor' depicted by the arrow in the middle. The corridor shows one agent in transit. That agent has a dashed outline to indicate is it is not active at the moment. Of course, this picture shows just about the simplest topology possible. We want to allow an arbitrary number of agents and agent spaces, arbitrary connectivity between different agent spaces, and an arbitrary number of concurrent agents in a corridor. Also note that with arbitrary connectivity we mean that a set of agent spaces doesn't have to be fully connected. There are very valid reasons for not connecting every space to every other space. It might be advantageous to partition a large cluster of agent spaces into work spaces and transit spaces. Work spaces allow for many concurrent agents to do serious work, while transit spaces just provide connectivity to many different other spaces, providing dispatching of agents, and for instance security features.

It is fairly logical to equip an agent space with non-mobile services, possibly implemented as (non-mobile) agents, that allow access to machine specific resources and/or other systems, including databases, legacy systems, and third party libraries installed on the machine.

Mobile agent
A mobile agent is an agent for which it is possible to move an active instance of the agent to another process: another agent space. It depends on the target process how difficult that is. It might be fairly easy when the process resides on the same machine, but pretty difficult if the other process is on a different machine, of a different type and running a different operating system. We examine in **Error! Reference source not found.** how to implement a mobile agent and using what technology.

5.4.1 Interaction model

Mobile agents obviously supports the same remote interaction models as the normal agent paradigm does (asynchronous agent to agent, and possibly synchronous agent to remote object), but because of the mobility, there are several other ones. The most important one we shown below:

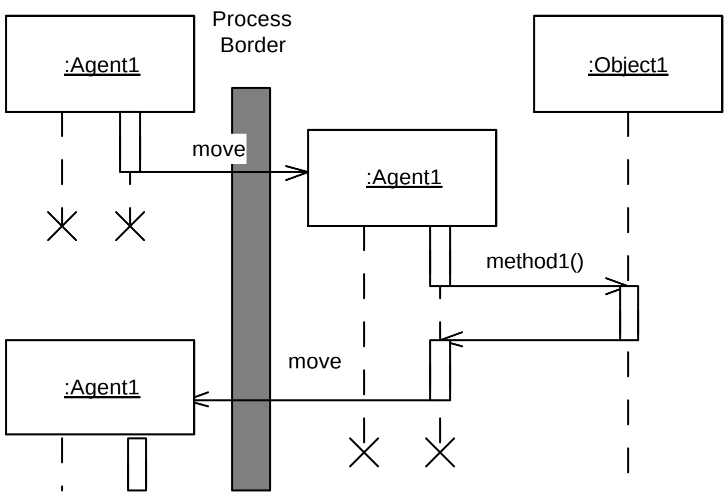


Figure 40: Mobile agent interaction.

You could see this as the object oriented, asynchronous version of remote evaluation. We don't send just code, but an object, thus state as well. The agent is able to autonomously interact with objects (or other agents) locally. Depending on what the agent does with the results of that local interaction, this could be vastly more efficient than doing the same thing through a remote interface.

Also, the originating machine doesn’t need to be connected to the other machine while the mobile agent does its work. It effectively delegated work to an agent on the other machine that wasn’t there before.

5.4.2 The difference

What makes this interaction model so different from the other models we’ve seen in the other paradigms? Well, it allows us to transfer the control of a computation to another machine. This, combined with the other interaction models, allows us to full control over the way processing *and* data flow around the system. At each point in a computation we can choose either to bring the data to us, or to bring the processing to the data.

We illustrate this using the distributed computation we’ve shown in 5.1.4. Recall we had three pieces of a computation A, B, C, and A was on the first computer, and C on A second, and B on either. The choice was where to put B. If we make this into an object model, we create three different classes, Object1, Object2, and Obect3, with a method A(), B(), and C(), respectively. Now, we need a fourth object to initiate the computation, to string the three elements together. We’ve argued in 4.10.5 that this activation ultimately has to come from some form of animate object. To make this explicit, we will make this object an agent.

Note that we made a choice in how we placed the pieces of the computation on different objects. The way we’ve done it is certainly not the only way to place three pieces of a computation can be assigned to different objects, but this way is the most illustrative.

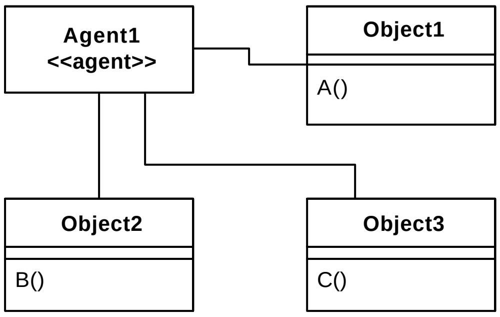


Figure 41: Three objects with each a piece of a computation.

Now, again the example says A on one computer, C on another, and B on either. The computation is starts with A, so we assume that Agent1 resides there as well. First, let’s put B on computer1. The object interaction diagram looks like this:

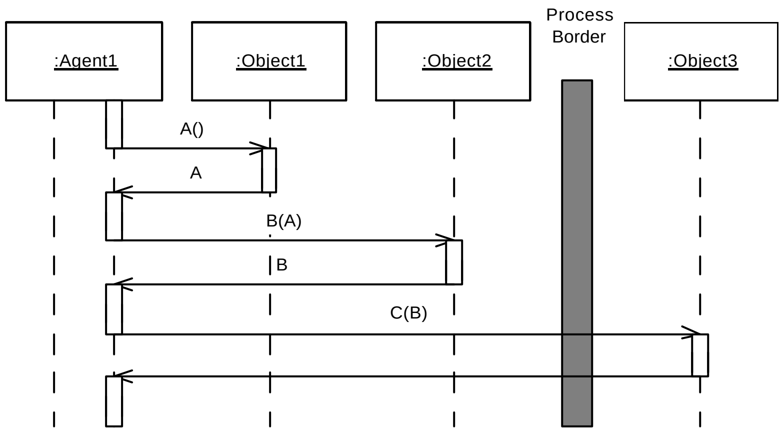


Figure 42: Distributed computation, B on 1.

Now, looks nice and simple. Nothing wrong, nice clean design. However, recall that the B was rather large, and we would prefer to have C(B) execute on the other machine. Assuming we have a suitable object B on the second machine, a simplistic attempt would be that we do it like this:

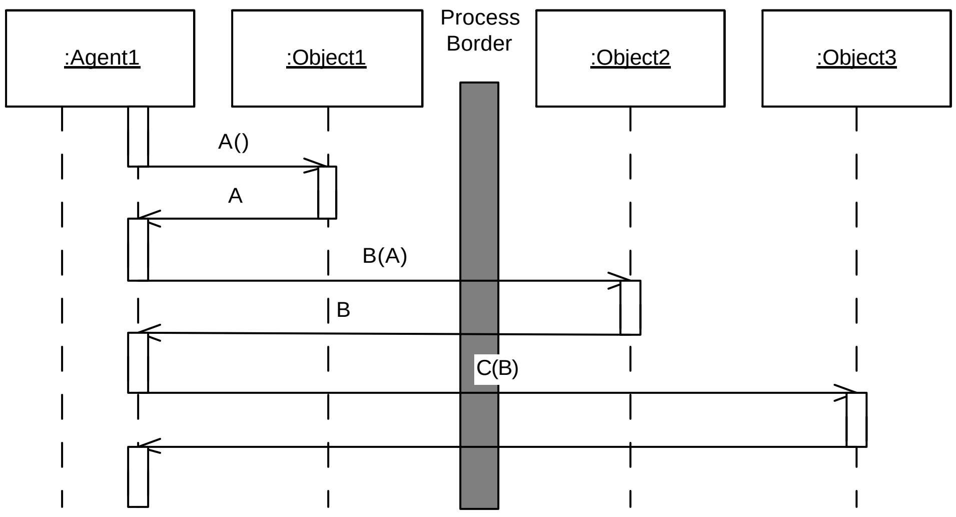


Figure 43: Distributed computation, B on 2.

Unfortunately, now the situation is most probably much worse. Sure, the first process is not taking care of the computation of B itself, but there are two extra data flows, A to computer2 and B to computer1. As shown, there is a needless flow of B back and forth. What do we do about this? Well, we can redesign Object2, and have it compute BC instead of just B.

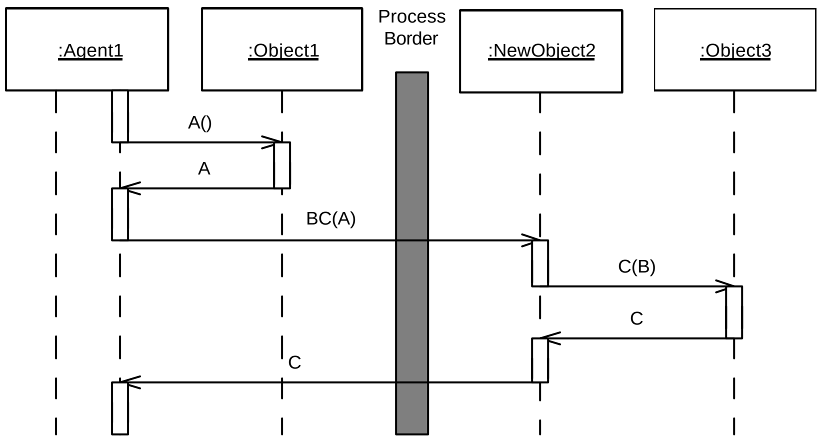


Figure 44: Distributed computation: B redesigned.

Now, we have what we want, the most optimal situation. It did however, require a design change. By itself nothing bad, but this change was necessary just because the initiating party wants to do a computation this way. If there are many different objects or methods, we quickly clutter computer 2 with a lot of modified objects, and a lot of methods. We would have Object2 compute BCD, DG, B*10+2G, etc, whatever we cook up for a kind of computation on computer 1. Is there an alternative? Yes, and its called mobile agents:

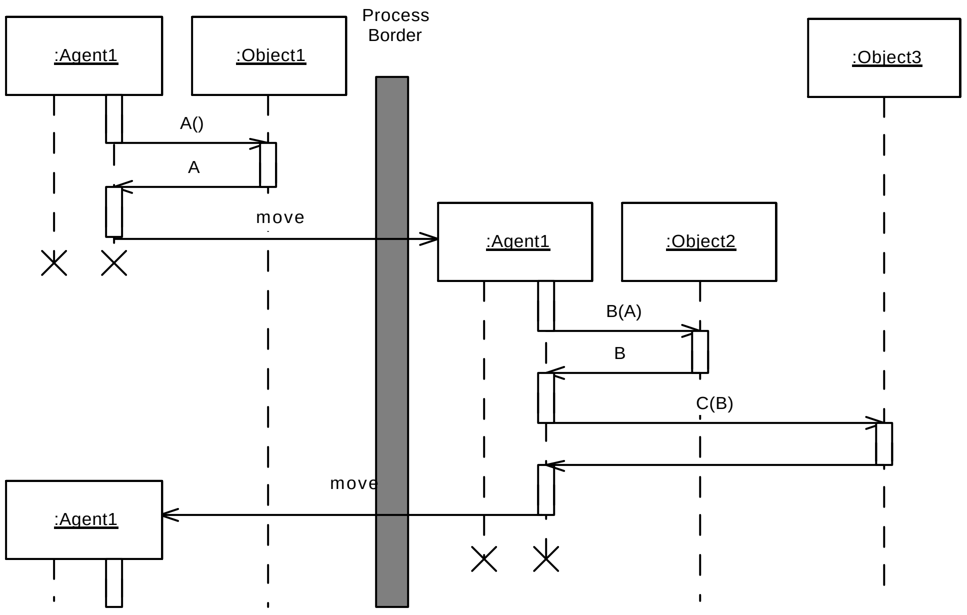


Figure 45: Distributed computation, mobile agent style.

control

We see now we have the same interaction as the previous example, but without a redesign. The difference now is that we move the *control* of the

computation to the other machine, instead of just the pieces of the computation. The picture shows also Object2 at the same level with agent, which could either mean the agent brought it from Computer 1, or even that it created it locally on computer 2.
Of course, for this to work, the object driving the computation of course has to be a mobile agent. But, the use of a mobile agent here means you can change the distribution of the computation by a simple code change in the agent, namely where you put the move instructions, instead of a possibly drastic design change of the objects involved on the other side. We could even defer the move decision to run-time if we wanted. And note that we don't have to do the move.

Finally, can we do this with mobile objects instead of mobile agents? To some degree, yes. If we equip the remote computer with a generic mechanism that allows you to send it an object it will activate and send back, you get a synchronous, object oriented version of remote evaluation. By its very nature, such a mechanism and such an object are very similar to agent spaces and agents. However, it still not as clean. The reason for this is that a normal object cannot move itself to another machine, because to move it, it needs to be inactive, but to move itself, it needs to be active. That's a classic catch-22. Now, it could send a *copy* of itself, but you can imagine that quickly gets messy. Other options result in different designs that work some scenarios, but we do not believe any would be anywhere as simple as mobile agents. Mobile agents get around the move dilemma because their autonomy allows them to support an asynchronous interaction model, in which they can return control to the caller and then issue the move.

5.4.3 Advantages

We don't pretend mobile agents are the silver bullet solution to all distributed computing applications. We do believe mobile agents provide some unique advantages that make it a worthy candidate solution for a distributed application architecture. To help such a decision we now look at some of the most important advantages of mobile agents, which the application architect will have to wager against the disadvantages presented thereafter. It is impossible to make a complete list of both, since any specific situation might make might make a minor plus into the big difference, or a small drawback into an insurmountable problem.

Minimal network traffic
A mobile agent is able to access resources locally, which can improve system performance and reduce network load.

Network disconnection
An agent space does not have to be permanently connected to the network. You can connect, initialise some agents and disconnect again. Connect again later (possibly at another location) to see the results (if any).

Transport transparency
The mobile agents paradigm provides transport transparency, that is, it hides how the transport of agents is implemented. Interesting enough, it

does so without requiring location transparency, thus without hiding where data resides

Simple services
Because a mobile agent can access things locally, we can keep a server interface simple. There is no need to smarten up the server to allow for more efficient client access. A mobile agent smartens up a server ‘on the fly’.

Minimal code base
Once an agent space is present, we can move in any code to a machine on an as-needed basis. Agents and related objects can be moved in once the need arises.

Late tuning
Tuning the distributed application to the network can be done at the last moment, not affecting the design of the application, and it could even be done on the fly.

Parallel processing
Agents execute asynchronously, so different agents dispatched to different locations execute in parallel.

5.4.4 Disadvantages

Mobile code requirements
The requirement for the agent to be mobile puts quite some stress on the code the agent is implemented in. It cannot be just any odd programming language or machine language because that might not run on the destination space.

Remedies: section 5.3.3 describes requirements for mobile code as well as some existing examples.

Mobile state requirements
The mobility of the state poses the same difficulties, and thus possibly the same limitations to the state type and values as those of the parameters of distributed object methods.

Remedies: the obvious solutions are the same as those for distributed object methods. If an agents is viewed (and implemented) as an object, object serialisation could help.

Agent space footprint

footprint

The *footprint* of an application is the amount of space taken up by the application on disk and / or memory. The code to move and run agents increases the footprint of (non-functional) application code, which could be prohibitive.

Remedies: For devices (i.e. handhelds, hardware with embedded chips) for which this is a problem, it is probably best to avoid having it host agents.

But then, the only thing need up front *is* an agent space, which means we can bring in new code on the fly which does not have to remain installed. If it is still prohibitive, a device could participate with an agent based application having it connecting it to a full-blown machine near the device, for instance have a PDA connect to a PC.

Security
Security for distributed applications is already pretty challenging. Agents increase the complexity because both the agent space could be threatened by a malicious agent and an agent could risk being dissected by a malevolent agent space. Receiving an agent on your machine sounds surprisingly lot like a having a virus run on your machine.

Remedies: The security of mobile agents has been extensively covered in the literature [14][29]. For instance digital signatures, authorisation and encryption could help here, like with many other security issues. The risk of malicious agents can be reduced by choosing an agent implementation language that is interpreted and with restricted functionality and / or build in security mechanisms, such as a scripting language or Java. Also, the security is only very troubling if we cannot trust the code of the agent. If we can, like in corporate network settings or because the agents are an integral part of the application, it becomes a lot less worrying.

5.5 Mobile agent architectures

We now want to show some ways the mobile agents can be used as the base architecture for a distributed application. We use class diagrams to display the different topologies. Stereotypes on the classes show the different roles in the architecture. We use generic names like Agent1, Agent2 to show different classes of the same stereotype.

We have included normal and dashed association lines in the diagram to show the difference between an architectural and functional association, respectively. That is, the solid lines show the basis of the architecture, while the dashed lines are just to indicate a possible association based on functionality.

All the architectures we show have the same ‘Front-end’ class in common. This could be either a front-end user application or a ‘server’, so serving as the outside boundary of the system.

Service agents
The first architecture we show is not utilising mobile agents yet, only normal agents. It serves as a good reference and starting point for the other architectures.

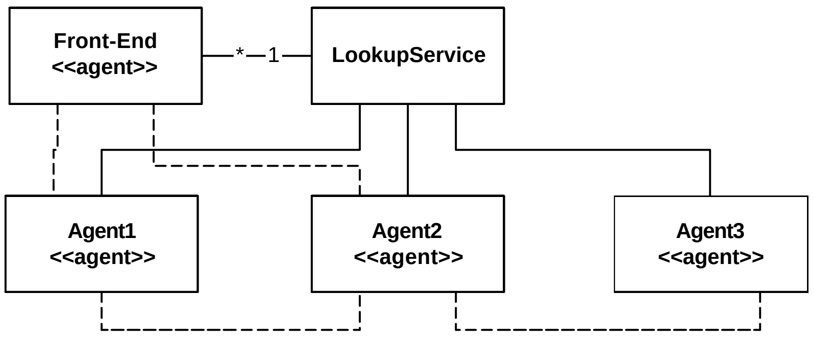


Figure 46: Service agents.

This is a basic distributed agents architecture each machine in the system is equipped with one or more permanent, service-like agents. The Front-end is also an agent (Each user would have its own instance of an agent) to highlight the inter-agent interaction that can take place. The agents would all use a lookup service to find each other and initiate the communication.

This is sort of the agent version of client-server, which thus support for delegation and a clean model for inter-server communication.

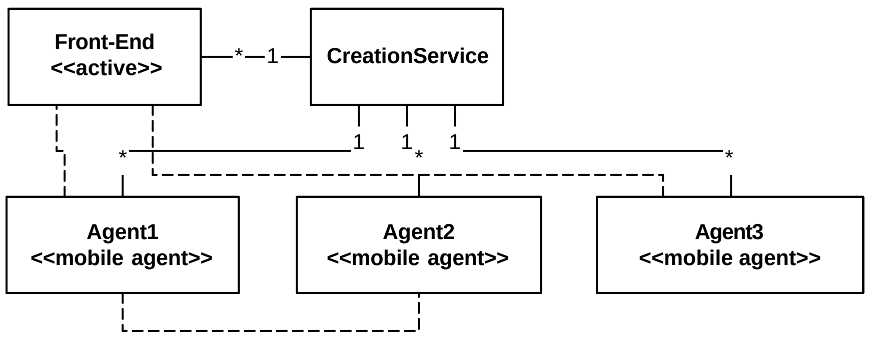


Figure 47: Mobile task agents.

Now, picture looks the same, but the roles are slightly different. We now have creation service instead of a lookup service, so we now have a front-end that is no longer an agent, but simply a shell around a particular agent space. An agent who wants to interact with the front-end (so possibly the user) could do so by directly communicating with the user. Each of the agents is thus truly an agent, completely autonomous and equipped with everything it needs to do its job. It is sort of like a mini-application by itself. The 'Front-End' object is allowed to disconnect from the network while the agents are roaming around and doing their work. One agent creating another can result in parallel execution.

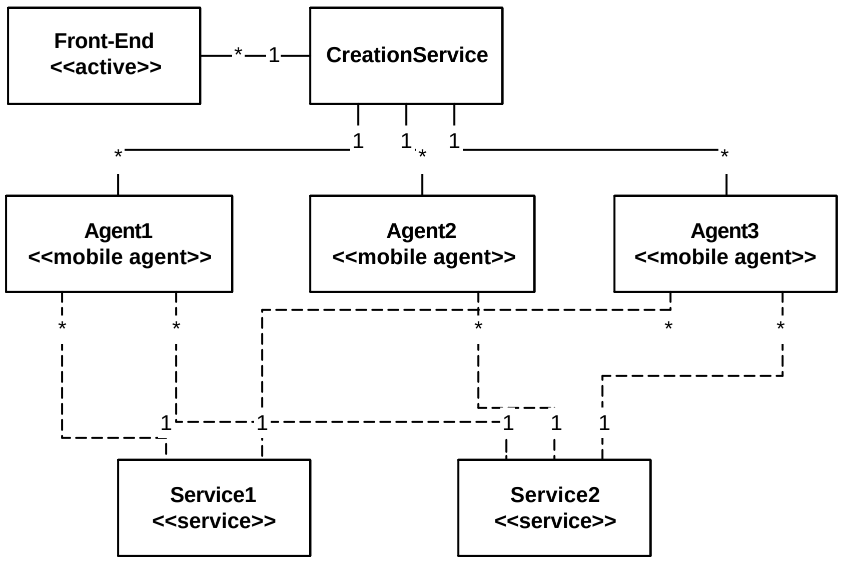


Figure 48: Mobile task agents and services.

This architecture extends the previous one by the explicit inclusion of services, or servers. Unlike the first diagram, these objects do not need to be agents, because the mobile agents can turn them into specific agents on the fly. By separating out the service functionality, we can make the mobile agents leaner, and we can better focus on their task. This could be a very typical enterprise application.

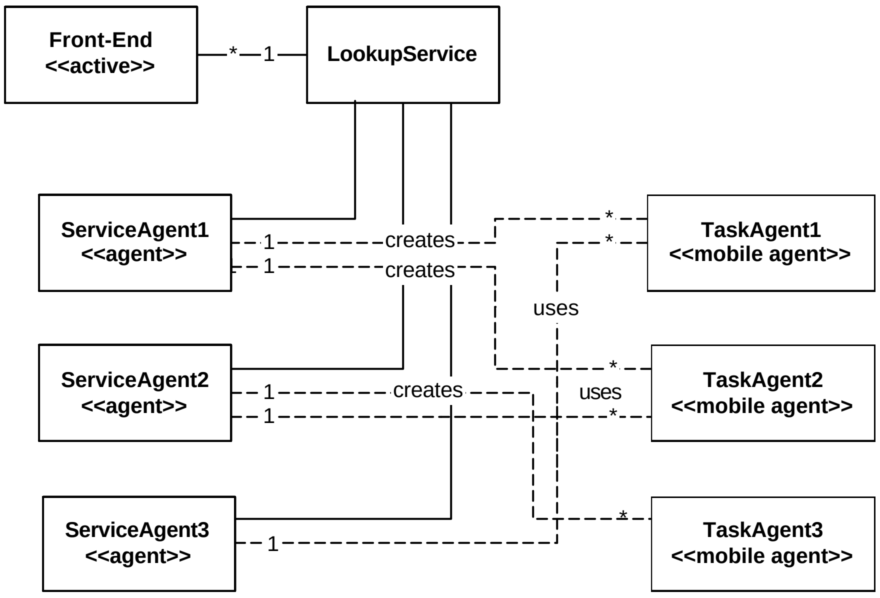


Figure 49: Service agents and mobile task agents.

Finally, we can have both service agents and task agents. Now, we only allow agents to create mobile agents, which in turn could use other service agents. This reduces the possible control distribution somewhat, but we can have the Service agents as mediators, and for instance provide purely synchronous interaction to the front-end. We can also trim down the mobile agents even further, because they don’t need any client interaction code. They just know how to talk to their ‘master’ service agent.

5.6 Existing mobile agent technology

Although not as plentiful as some of the more established paradigms described in 5.3, there are several technologies that directly support mobile agents. We briefly discuss several below:

TeleScript
TeleScript, by general magic [37], was one of the first mobile agent implementations. TeleScript is an agent scripting language that runs in TeleScript engine, the agent space for TeleScript agents. One of the unique aspects of TeleScript is that its execution can be stopped at any point in time and transferred to another engine. The programmer only issues a ‘moveTo’ instruction, and does not have to take any other special care. TeleScript is a proprietary language, but general magic recently released Odyssey, a Java implementation with much the same features.

Aglets
Aglets are IBM Japans implementation of mobile agents. ‘Aglet’ is a concoction of ‘agent’ and ‘applet’, which exemplifies that agents are like little applications and that aglets are implemented in Java[14]. Aglets run in an Aglet server, but unlike TeleScript, Aglets are implemented in a normal programming language. The aglet server also comes with an extensive security mechanism that makes it safe to run untrusted aglets on your server.

CORBA mobile agent facility
The OMG [39], the organisation that defines and maintains the CORBA standard, is attempting to standardise (some aspects) of agent technology. This effort, appropriately called MASIF, (Mobile Agent System Interoperability Facilities) [40], defines how agents could be managed and transferred in a CORBA setting, and also standardises the naming of agent systems, locations, and the agents themselves. This way, agents written in the same language, so for instance, Odyssey agents and Aglets, could move themselves to servers that implements the MASIF specification. Effectively, this means an Odyssey server could run and manage aglets, and vice versa. Together with a CORBA ORB, this gives a very powerful standard to build open (non vendor-specific) agent systems.

ObjectSpace Voyager
Voyager is a family of products with build-in support for mobile agents [41][42]. The simplest version is what ObjectSpace likes to call an agent-enhanced orb. That is, the Voyager orb is not just able to handle request for remote objects, but can also receive agents that are instantiated for remote execution. Top of the line is the Voyager application server, which in addition supports Enterprise Java Beans, Distributed transactions, centralised server management and other advanced application server features.

5.7 Possible applications

In 5.4.3 we described the advantages of mobile agents. Here describe how those advantages can be used to create some very desirable features available for applications based on mobile agents. We then elaborate on the applications, in two mayor settings: Mobile agents on the Internet, and Enterprise applications.

Parallel processing
Agents provide quite a natural metaphor for parallel processing: A co-ordinating agent spawns many copies of a task agent that works on part of the computation. If each of these agents is a mobile agent, we enable true massively parallel processing.

[14] Curiously enough, an aglet is also the hard plastic bit at the end of a shoelace.

Intelligent messaging
Mobile agents can be seen as a form of intelligent messaging, the agent being the message. The autonomy of the agent could enable the message to find its own optimal route, wait until a machine is up or connected to the network, or dynamically retrieve data as the receiver requests it. This is also an interesting way you could create a 24x7 illusion for a not-24x7 system. Doing some sort of transaction means sending the intelligent message. For the sender this is 'end of story', but if the target system is unavailable, the message will try delivering itself until it is received.

Load balancing
In computing, load balancing means a set of machines that can do the same tasks co-ordinate to make sure each is taking a roughly equal share of the joint load. This ensures optimal response time for all users of the system.

Suppose a mobile agent can ask the space it is running in how busy it is, and can pose the same question to other agent spaces. If another space is much less busy, the agent can decide to move to the less busy server. This is highly decentralised, and thus highly scalable option for load balancing. Alternatively, the space could be made responsible for balancing its load with other, 'close' servers. Close should be taken quite liberally, and could be defined by the system administrator, or for instance determined by bandwidth between the servers. In any case, we mean it would not have to co-ordinate with all servers in the cluster to perform successful load balancing.

Fail-over
Losing a transaction because of some failure of some component can be quite disastrous. Fail-over refers to a set of techniques to overcome this failure by having some other component take over in the event of failure. Mobile agents can provide a method for overcoming server failure. Upon creation, an agent spawns a copy of itself and sends it to another space on another server. The copy stays inactive and receives 'heartbeats' of the active agents. When the original agent dies because of server failure, the dormant agent misses the heartbeats of that agent, and wakes up, becoming active, and probably making a copy of itself and sending it to another server.

Of course, state updates have to be send to the inactive copy. A simple way would be to send a new copy each time and destroy the old one. A slightly more advanced way is to use normal inter-agent communication to update the copy.

Semantic routing
Semantic routing is more or less the result of the network awareness argument above. The mobility of the agents allows us to define an optimal execution path for a particular agent, based on the data requirements of the agent. For each piece of the computation, we can choose to send the agent to another place, or to retrieve the data remotely. Either the programmer is in control of this, issuing 'move agent' instructions as part of the computation, or a smart mechanism build into the agent spaces knows when and where to move the agent.

Semantic routing could be very valuable for applications that need to run on a set of machine which have wide geographical dispersion and / or big differences in network bandwidth in between machines.

5.7.1 Mobile agents on the Internet

There are many articles out there that describe mobile agents in relation with e-commerce. They propose to allow user agents with a task such as 'get me a cheap flight to Hawaii in June' to travel to any server on the Internet and hop from server from server and finally returning to the user with a set of possible flights or even with the cheapest flight already purchased. While this is definitely possible, there are many issues with it, and it is both to too limited and too generous.

Too limited because, as we argued, there are many other possible applications of mobile agents, most notable in the area of enterprise computing.

Too generous because allowing arbitrary pieces of code to roam around a network puts huge strains on the security system, and even extends into the legal area. Suppose a server is compromised by some agent. Who is responsible: the entity owning the machine? The entity who wrote the agent? What about the server the agent came from? And what if the origin of the agent is unknown or forged?

We can almost hear the lawyers rubbing their hands together after the first rampaging agent on the loose. Because of all these security implications, many of those papers focus on how to overcome those, which is unfortunate, because they make mobile agents seem unnecessarily complex. Also, for such a service to succeed, you need an existing infrastructure of agents spaces on Internet servers. There is no clear incentive for e-commerce providers to add such a server to their Internet offerings. They would prefer it if the client, and preferably in person, visits them and only them, instead of them and a number of competitors. And even if they would the security implications would make them think twice.

Does this mean we should forget about mobile agents and e-commerce? Absolutely not. Let's take one step back: why should a client want this type of services: because of the comparison, the asynchrony, and to side-step bandwidth limitations. But to do that, the agent doesn't really need to travel to each server, but just to a permanent server on the Internet. It could engage in 'normal' distributed computing interaction with the other servers, a most obvious choice being the same method a real person uses: the web interface. This means there is a clear opportunity for third parties to start hosting client agents. Because they provide the agents, and the agents only travel between the client and the hosting service, the security requirements are much less. An obvious candidate for a client agent hosting service would be an Internet portal.

If we forego the comparison aspect a bit, it is also interesting for an e-commerce provider to allow mobile agents travelling between the client and

Internet servers. Such agents could offer similar asynchronous services and / or provide customisation and high degrees of interaction.

Summarising, mobile agents definitely have a future for the client side of e-commerce. We believe that by restricting the mobility of the agents you can provide many of the advantages without all of the issues associated with arbitrary mobile agents.

5.7.2 Enterprise scale applications

The ever-growing need for more and more integrated information within companies warrants the creation of ever larger, enterprise scale applications. A core concept in this area is 'business logic'. A piece of business logic is code encapsulating a piece of business functionality for the enterprise. For a bank this could 'withdraw money form account' or' create savings account'. For a book retailer this could be 'get list of books in stock' or 'ship order'. A current trend is to put this business logic in a middle tier layer, that is a set of network servers that host and expose this business logic in some way. The enterprise application consists then of simple applications on the client side that employ the business logic to get the job done, or, another trend, of a web server that presents the application on web pages so the application can be used through a web browser.

Existing applications often employ middleware-like application servers and CORBA for robustness and structure. We think mobile agents could prove to be an interesting option in this area. The 'mobile' of the mobile agents can be use to provide application server-like abilities and the 'agent' part a good metaphor for structuring the application in agents. For some ideas of how an enterprise application could be structured using agents, see the examples in chapter 2.

The 'mobile agents as application server' idea warrants some further explanation. We believe it could be extremely interesting to consider a mobile agents based application server. A cluster of machines with smart agents spaces that are able to do load balancing, fail-over and request routing as described above could be made into a pretty robust and highly scalable application server. It could be 'the TCP of application servers'. This is a pretty bold statement, but let us explain. We do not mean that we think a mobile agents application server would be nearly as widely used or standardised as TCP is. What we do mean is that it could have the same robustness and efficiency as that which TCP made so popular. A TCP network is able to route requests as efficiently as possible, is able to recover from data loss, and as network use changes and servers are added or removed, the network automatically 'reconfigures' itself to the new situation. There is no centralised control, which means failure of some machine or set of machines hardly ever results in failure of the network.

It is this kind of behaviour, but then at the application level, that we think can be achieved using mobile agents. Service agents (holding business logic) can be strategically distributed across the network. As the network environment changes the served agents can be moved or copied to better serve the new situation. Individual requests can be created as an agent, and be subject to semantic routing, meaning optimal execution efficiency. Components requiring fail-over can use a strategy such as described in the beginning of this section. Like with TCP, there is no centralised control, resulting in a very robust and very scalable system.

We would very much like to experiment building a system using these ideas, although this thesis is not the place, as building such a system is quite time intensive and complicated, and we lack the time.

5.8 Conclusion

We have described several different paradigms for the design of distributed applications. Each paradigm improves on the previous by giving the designer more flexibility in partitioning the logic over different components.

The first paradigm, *distributed objects*, basically requires your design to have a static distribution of objects in the system. Each of the machines in the system needs to be pre-equipped with the code for the classes it needs, and inter-machine communication is limited to primitive types, although that might take the form of a few specialised objects if your try hard. By itself, there is nothing wrong with all this, and many distributed applications are build using this paradigm, like ‘traditional’ client-server applications.

The *agents* paradigm still requires a static distribution of objects in the system, but it allows for a more dynamic interaction between the different pieces, by enabling a structured form of asynchrony. It also provides a better metaphor for the inherently passive ‘server’ concept.

Now, *mobile code*. Mobile code allows a more dynamic distribution of objects. It is most often employed in a client-server setting, to equip a client with new functionality. Once the client has a basic framework to support mobile code, new functionality can be send to the client as needed. For the web, this ‘framework’ is thus the web browser supporting scripting languages (JavaScript, VBScript) and applets (Java applets, ActiveX controls). By itself, the allocation of objects remains static, but the designer has more freedom of where to put the objects, and is better able to take advantage of the client’s processing power.

A big change occurs if we put distributed objects and mobile code together: we get *mobile objects*, we are able to send instances of objects (rather than code to create a new instance) to other machines on the network. The distribution of objects becomes completely dynamic. Sure, we can simulate mobile objects to some degree using the other paradigms, but the underlying technology cannot directly support it. A design using mobile objects thus consists of at least a few objects that stay put on each machine (or in each process) that participates in the application, and a bunch of objects that roam freely between those objects.

At the very end, we get *mobile agents*. Mobile agents allow us to not only move around objects, but the control of a computation as well. This creates a solid foundation for applications that need to be very network aware, and allows for disconnecting clients, delegating tasks, and parallel processing. Servers can be kept simple as they are enhanced on the fly by an agent visiting them.

6

CHAPTER 6 Implementing agents and mobile agents

In the previous chapters we have shown many possible applications of agents and mobile agents. But how do you go about implementing those, and what technologies could you use to support you? In this chapter we show some ways (mobile) agents can be built, and point out some issues and considerations.

Implementing a basic agent is not that hard. It might take some effort to implement good sensor and effector logic, such as reading out a hardware device or sending emails. But all that kind of logic is inherently domain specific, and is not really necessary just because you want your system agent-based. Therefore, we restrict ourselves to explaining how to get basic agent behaviour, how to create a shell in which an agent can be placed.

The code examples in this chapter use Java as their programming language, mainly because Java is widely used nowadays, and has nice built-in threading functionality. It is definitely possible to use the same kind of implementation in other general purpose programming languages.

6.1 Agent implementation strategies

Because agents are such a broad terrain, and the agent concept is so flexible, there is no single ‘right’ way of implementing agents or mobile agents. We want to distinguish three different basic strategies for implementing (mobile) agents: scripted, based on a framework, and DIY, or: Do It Yourself. These options range from relatively simple and fast to implement, to more time-consuming but very powerful.

Which strategy is the most suitable for a particular situation is hard to say. Generally, if there is a script language or framework that suits your needs, it is the preferred choice. However, if it means making heavy compromises you should not be hesitant to bite the bullet. The DIY section should give a good impression on the advantages and perils of a DIY implementation, as well as some pointers to non-agent technology that could help such an implementation.

6.1.1 Scripted agents

scripted agent

The simplest is a scripted agent. Such an agent can be constructed by writing code for the agent in a dedicated, often proprietary agent language. This code is then interpreted in a runtime environment for that agent language.

A simple, imaginary example would be something like:

```
STATE:
News = HTTPDATASTREAM(www.reuters.com/news)

CODE:
IF News.changed() AND News.contains(“CATP”) THEN
   DISPLAYALERT( “Reuters has news on Cambridge”)
```

The idea is that this agent would display an alert message to the user as soon as a news page at Reuters is updated and contains news on Cambridge Technology Partners (CATP). To run this agent, the agent environment would first initialise the state of the agent (in this case the news stream) after which it would periodically run (interpret) the code piece of the agent.

The advantages of this kind of approach is that it is very fast and easy to build the agent, since you only have to create the ‘brain’ logic of the agent. A disadvantage is that if the agent language doesn’t allow you to sense the data you want, or take the actions you want, you are out of luck, or have to add that to the environment in some way.

Some not-agent applications also allow for this kind of ‘agents’ to enhance their functionality. For instance, Microsoft Outlook allows you to create ‘inbox assistants’ to sort your mail in separate folders. In this case, the agents are constructed using a GUI, but the idea is the same.

TeleScript of General Magic [37]is a good example of an agent scripting language. It is a generic, but proprietary language that allows you to create (mobile) agents.

6.1.2 Using a framework

framework

More powerful, but thus more complicated, is when you write an agent in a generic programming language such as C++ or Java, or in a more specialised language such as Lisp or Prolog. A *framework* is a set of pre-built components that can be used to build a certain style of applications. Current research shows a lot of frameworks and architectures for building agents (see [1]), so chances are there is one that will suit your desired application.

The main difference between a framework and scripted agents is that when using a framework, you use a ‘no limits’ general programming language, often the same as that in which the framework itself was written.

Even if there is no existing framework that can help, it is probably a good idea to create a basic, supporting framework for the agent application. We also took that approach when developing our prototype.

We listed some mobile agent technologies in section 5.6. Two of these, IBM’s Aglets [38] and ObjectSpace Voyager [42] fall into this framework category. Aglets are an agent specific, free framework for mobile agents, with features to host untrusted agents and even an agent locator application. Voyager on the other hand is a full-fledged commercial product, referred to as an ‘agent enhanced object request broker, which can be used to build distributed applications that include (mobile) agents.

6.1.3 DIY

do it yourself

As said: Do It Yourself. Build everything yourself, using a general programming language, possibly using some existing, not agent specific, technology to get started. This is of course by far the hardest, longest to develop way of developing agents, but it might be the way if none of the existing frameworks suits your needs.

It is also possible you have just found that there are a few components of a system that need agent-like behaviour, but you are not really building an agent-based application. We referred to this earlier as a pro-active system. If the behaviour of those agents is not that complicated you can build those agents using a few elementary programming techniques. In any case, it is probably a good idea to look at other agent systems and carefully examine the underlying idea’s and technologies. There will no doubt be a few that can be very helpful in supporting your implementation.

The basic problem that you need to solve when doing a DIY agent implementation is creating autonomy. Agent frameworks would already have taken care of this for you, but here you are on your own. Creating autonomy is not explained in a few lines, so we defer this to a separate section: 6.3.

Further, if the agents need to be mobile, as described in chapter 5, we describe in section **Error! Reference source not found.** how this can be accomplished.

Just because there is no agent framework or agent scripting language to support you doesn’t mean your are on your own completely. Many technologies could support building an agent application. We review a few of them:

Object oriented programming languages
The most basic option. We have already shown in chapter 4 that agents fit in an object-oriented view of software. Such a model is easily translated into an object oriented programming language. Agents generally are complex

entities, which means it is probably wise to divide its logic over several objects. In other words, and agent is a component (see 4.9.2). One object should represent the agent to the rest of the system, and harnesses the autonomy. This object is then a ‘façade’, (see also the façade pattern in [43]) and uses other objects for the implementation of for instance sensors and effectors. There’s more on this in section 6.2.

Distributed objects
Many agent applications need to be distributed across several machines. Then distributed object technology (see 5.3.1) can help. An additional advantage is that these technologies are equipped with a lookup mechanism for finding remote objects. Such a mechanism quite naturally poses as a way to find specific agents in the system.

Messaging systems
Autonomy means asynchrony. Therefore, many agent systems require some sort of messaging systems to enable inter-agent communication. This could be accomplished using some sort of middleware such as MQ-Series, or could for instance based on some implementation of JMS, the Java Messaging System API.

We envision it might also be possible equip each agent with an SMTP mailbox, or on a single system, a simple home-made message queue. One of the fun things about agents is that there is a lot of flexibility in how you add such capabilities to a system. You can make each agent able to talk using some messaging technology, make it a system service, or create a messaging agent that relays communication to other agents. This more or less forebodes 6.2.2.

Application servers
An application server can provide quite a natural hosting environment for agents. They provide a solid runtime framework for multi-user (or multi-agent) applications, and often provide base functionality an agent might need, such as database access and logging. A disadvantage might be that you are tied to a particular threading model. You need to rely on mechanisms within the application server to provide the agents with their autonomy. It is interesting to note that for instance Lotus Domino [43] uses the term ‘agent’ for its basic processing components. Whether those ‘agents’ fit under our definition of agent is a bit dubious. Sometimes such an agent can really function as an autonomous component, but it other settings it is just activated when something has to happen, for instance the display of a web page.

6.2 Basic agent models

In chapter 4, we mainly shown how to model using agents, but to create an agent application from scratch, you need to model the agents themselves as well. Recall from chapter 2 we defined an agent to have five basic components, sensors, effectors, a brain, a heart and memory. How do you put those components in an object model?

6.2.1 Heart, brain, and memory

In order to perform its work, the agent has to execute a continuous cycle to sense, reason and effect. We call this cycle the 'beat' cycle, reflecting that that cycle executes in a single beat of the agent's heart.

Simple agent
For a simple agent, this looks like this:

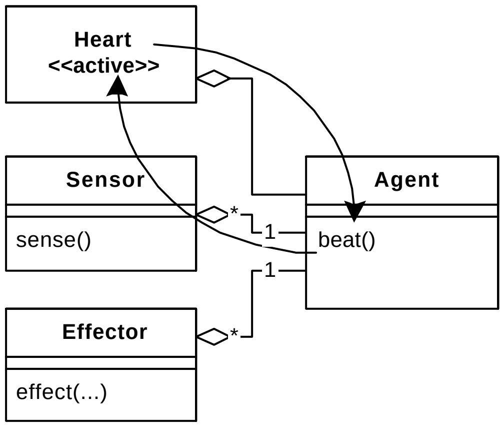


Figure 50: A simple agent.

From this picture, the brain and memory components are suspiciously missing. For this model, the brain is thought to be an integral part of the agent, inside the beat() method. The absence of memory indicates that this is a purely reactive agent. In a real agent, the sensors and effectors would be concrete instances of certain classes, and by no means would they need to have a common 'Sensor' or 'Effector' base class. This is done here just for illustrative purposes.

The arrows indicate the beat cycle, driven by the heart. You might be wondering how the heart object functions. In a simple case, the heart object can be a thread. There are a few other ways this might work, and we explain how in section 6.3.

Now, the 'beat' method. How this looks for this simple agent is best explained in a code example:

```
class Agent
{
Sensor1 _Sensor1;
Sensor2 _Sensor2;
…
Effector _Effector1;
Effector _Effector2;
…

void beat()
{
   String SensorOutput1 _Sensor1.sense();
```

```
    int SensorOutput2 _ Sensor2.sense();
    …

    if (SensorOutput1.equals(“Something”) ||
  (SensorOutput2 == 4)
    {
      _Effector1.effect(…);
    }

    if (SensorOutput2 > 17)
    {
      _Effector2.effect(…);
    }
    …

  }
  …
  }
```

As can be seen in this example, the ‘brain’ of this agent is formed by a set of conditional statements that trigger an effector. This example by far doesn’t show all possible options. It is entirely possible some sensors have other result values, more than one effector is triggered in one conditional, a case statement is more effective, conditionals are nested, etc.

A more complex agent

You can imagine that for a complex agent, the conditionals in the beat() method quickly become unwieldy, making them hard to understand and difficult to maintain. Also, those rules are fixed, so there is no way adaptation, or learning, can happen without the programmer’s intervention. With agents this complex, you want to create a separate brain object, capable of doing the reasoning. This kind of agent is probably an intelligent agent, and that brain would use some kind of AI technique, such as a neural network, the BDI formalism (3.2), or a predicate based deduction/induction method. This thesis it not the place to discuss or recommend those techniques, but the references section point to plenty of possibilities, especially [1] and [2]. In a picture, it looks like this:

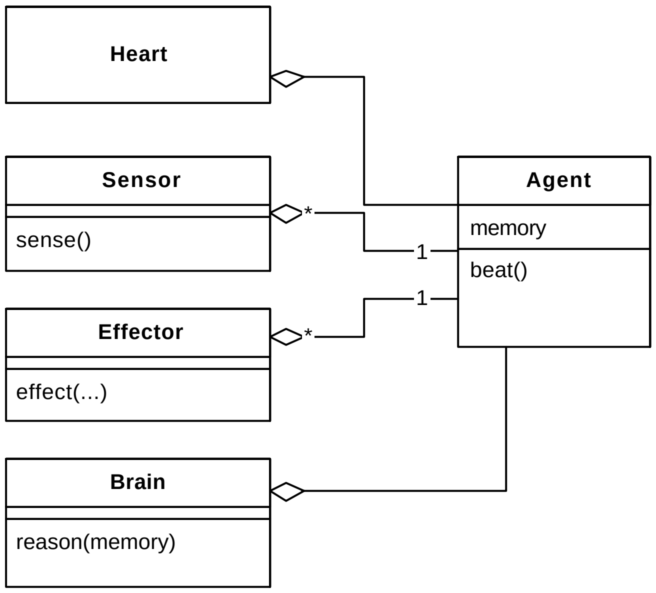


Figure 51: A more complex agent.

No mayor additions, just a brain object and the addition of memory as part of the agent's state. What specific form the memory takes is largely dictated by whatever components the brain uses to reason with. The role of the beat method now becomes somewhat different. Instead of being the brain of the agent, it is now responsible of transforming sensor input into memory updates, triggering the brain, and translating the output of the brain to effector activation. It no longer does any reasoning, put purely provides mapping.

It might be that the brain updates the memory. So if for instance the memory is a set of formulas in predicate logic, that set can be different after the brain has reasoned with it. A new beat method could look like this:

```
class Agent
{
Sensor1 _Sensor1;
Sensor2 _Sensor2;
…
Effector _Effector1;
Effector _Effector2;
…

double[] _Memory()

Brain _Brain;

void beat()
{
   String SensorOutput1 _Sensor1.sense();
   if SensorOutput1.equals(“Something”)
   {
```

```
            _Memory[0] = 0.45;
        }

        if SensorOutput1.equals(“Somethingelse”)
        {
            _Memory[0] = 0.9;
        }


        _Memory[1] = _ Sensor2.sense();

        _Brain.reason(_Memory);

        if (_Memory[10] > 0.9)
        {
            _Effector1.effect(…);
        }

        if (_Memory[11] > 0.9)
        {
            _Effector2.effect(…);
        }
    }
    …
    }
```

In this example, the memory consists of an array of floating point numbers. The brain that takes these numbers as input as for instance a neural network would. We assume here that the brain places its results in memory locations after 10. What effector should be activated is then determined by inspecting these memory locations. Again, this is just an example, and the memory might be a lot bigger, the activation conditions might be different, or several memory elements together determine the activation of one effector. However, if this mapping of memory values to effector activation becomes too complex, the brain isn’t doing enough reasoning on its own.

Adaptive agents
Now, adaptive, or learning agents. Adaptive agents require the brain to get a chance to modify itself. Although for some type of brain implementations, this might be an integral part of the reasoning process, it is useful to make the distinction.

Note that an agent might require that its sensors input includes the results of effector activation. However, to program this, it is a little easier to assume the effectors return the result of the activation, which is then stored in the memory of the agent. Otherwise, effectors have to double as sensors, and have to remember the result of the previous activation.

```
class Agent
{
Sensor1 _Sensor1;
Sensor2 _Sensor2;
…
Effector _Effector1;
Effector _Effector2;
…

double[] _Memory()

Brain _Brain;

void beat()
{
   String SensorOutput1 _Sensor1.sense();
   if SensorOutput1.equals(“Something”)
   {
      _Memory[0] = 0.45;
   }

   if SensorOutput1.equals(“Somethingelse”)
   {
      _Memory[0] = 0.9;
   }

   _Memory[1] = _ Sensor2.sense();

   _Brain.learn(_Memory);

   _Brain.reason(_Memory);

   if (_Memory[10] > 0.9)
   {
      _Memory[20] = _Effector1.effect(…);
   }

   if (_Memory[11] > 0.9)
   {
      _Memory[21] = _Effector2.effect(…);
   }

}
…
}
```

Here we see an additional call to a ‘learn’ method on the brain, and that memory after position 20 is updated to reflect the result of the activation. A tricky aspect to be aware of is that both the ‘learn’ and the ‘reason’ method might affect the memory, which could influence the learning process. If this undesirable, care must be taken that the learning process learns from an older copy of the memory not affected by reasoning.

6.2.2 Effectors and sensors

There are several ways an agent can relate to its sensors and effectors, and we want to show a few of the most common ones:

Integrated
Like the model presented in chapter 2, here the sensors and effectors are really part of the agent. However, sensors and effector *classes* can still be utilised by different agents, like this:

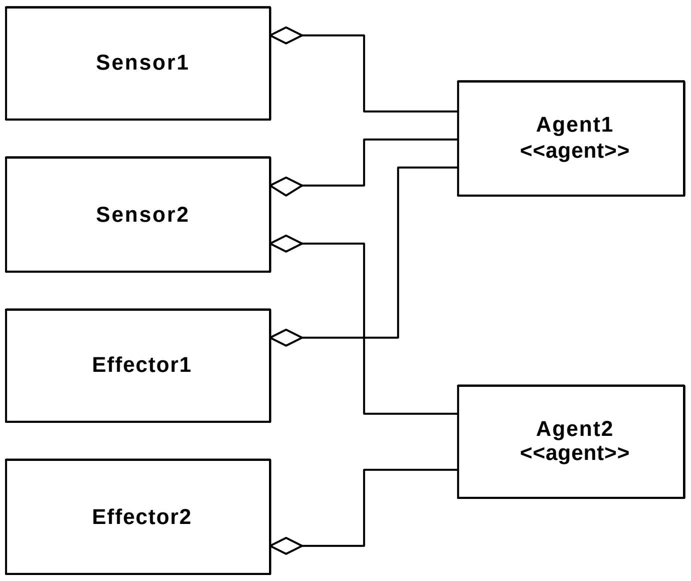


Figure 52: Sensors as part of the agent.

Shown are two agents: 'Agent1', using 'Sensor1', 'Sensor2' and 'Effector1', and 'Agent2', using 'Sensor2' and 'Effector2'. We use the same topology in the next examples as well. In this case, each agent has its own instance of the sensors and effectors. This model makes the agent fully autonomous in every sense, since it doesn't have to rely on anything else to do its job.

Sensors and effectors as services
It is also possible to think of those sensors and actions as passive 'services' in the system. That way, the services provide the functionality, and the agents are what make the whole thing 'tick'.

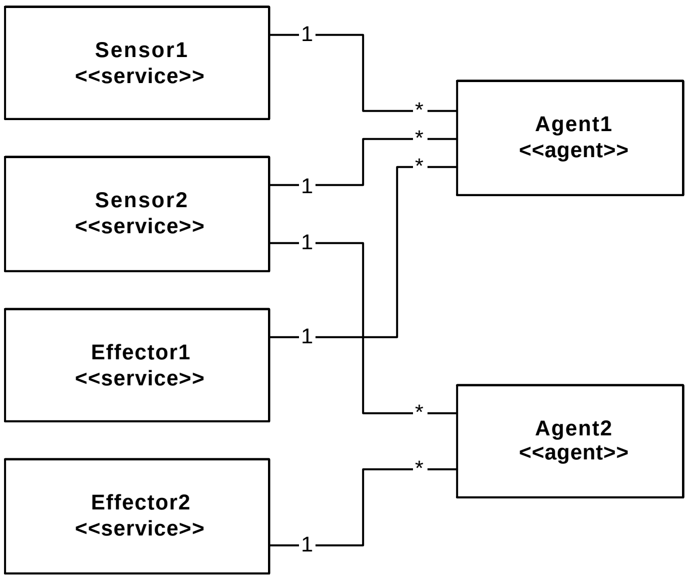


Figure 53: Sensors as services

The picture looks very similar to the first, but there are some fundamental changes. Now, the sensors and effectors are stereotyped as 'service' by which we mean they are a system-wide services, generally single-instance. One instance of such an object is constructed at start-up, and it stays there as long as the system is up, a bit like an in-system server. As a result, the sensors and effectors are no longer part of the agent, but part of the system, and many agents might relate to the same service. A design like this makes sense if the agents need to use resources that are inherently machine bound or require machine-wide management of scarce resources.

Agents using agents

A third option is to equip each agent with a limited action and/or sensors. The agents would then rely on each other to provide the system's functionality. Like this:

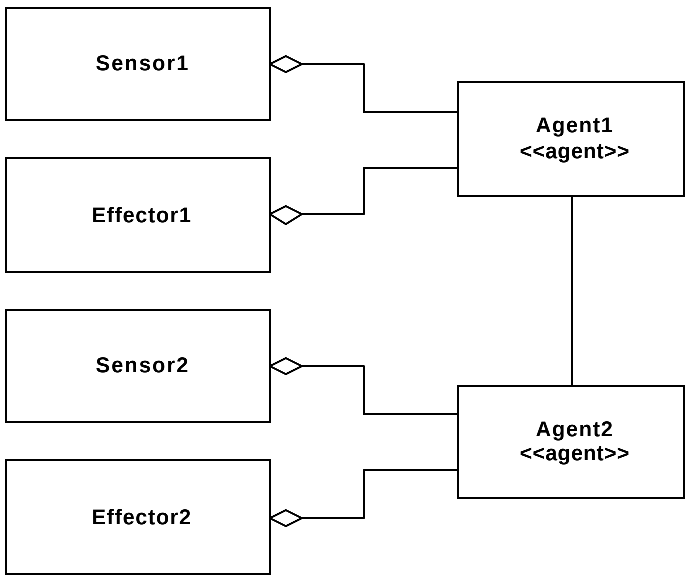


Figure 54: Agents using each other.

federation

Compared to the first picture there is only line of difference, Agent1 is not connected to sensor2, but to agent2. There is a substantial difference in concept however. This is a very primitive example of a *federation* of agents. 'Federation' is a relatively new term in software, and used for more or less autonomous computing devices that work together to provide a system that appears as one. In Sun's terms: 'The network is the computer'. A federation of agents is similar, but now the 'devices' are strictly software.

In this setting, if an agent cannot do something himself, he will delegate it to another agent. Which agent something is delegated to can even be decided at run-time. There are systems that use a 'blackboard' on which agents can place requests services. Other agents can look on the blackboard, and propose, even bid, to offer the service [45].

6.3 Creating autonomy

We think autonomy is the core of what agents are all about. In this section, we describe how this autonomy can be achieved in a computer system. If each agent resides on its own computer, this doesn't take any additional effort, since the computer is already autonomous, or animate, by itself. That is, the computer is able to operate by itself without any help. Sure, normally, it just sits there passively responding to your commands, but if you have a screensaver installed, you know that the computer will, by itself, decide that it hasn't been interacted with for a while, and enlighten the room with a nice animated aquarium[15]. So if the computer's sole purpose is running an agent, it can and will show autonomous behaviour.

It gets a little more difficult if the computer has to run more agents simultaneously. You need a way to distribute the control over the CPU in such a way that each agent is perceived to behave autonomous. Before we can explain what the options are, we need to explain a little about processes and threads.

6.3.1 Processes and threads

process

A running instance of a program on a computer is called a *process*. A process includes all the code and data of a program currently working with in memory. Double-clicking the Netscape icon on a Windows system starts Netscape Communicator, and at that moment Windows is running a Netscape Communicator process. Older operating system such as MS-DOS could only run one process at a time, but modern operating systems allow multiple, simultaneous processes. Those processes are *isolated*, meaning that they cannot normally read or write in each other's memory. This is important, because it means a process cannot accidentally or purposely affect the other processes, which could cause those processes to crash or malfunction.

On a multi-processor system, the computer might be running some processes truly in parallel, but generally, the operating system pretends it is running the processes in parallel by very quickly switching between all the currently running processes. This is referred to as multitasking

thread

Initially, a process has one *thread*. Thread is short for 'thread of execution'. A thread is a possible route through the code of a program. If, in slow motion, and on a separate monitor, you could see the code of the program your computer is running, the thread would be the continuous highlighting of the lines of code the computer was executing. If you don't do anything, to computer would probably be looping through a few lines, and if you press a key, you see the highlight jump out of the loop to process that keystroke, possibly update the screen, and continue in the loop.

Many operating systems allow you to start additional threads in a single process. This is very convenient, because, like your computer (pretending) to be running more than one process, the program can more easily do more than one thing at a time. For example, newer word processors check the spelling of your text while you type. If you have to program that using only a single thread, that is possible, but nasty. One reasonable way is after each keystroke to check if the word was completed, and if yes, check it for spelling. However, checking the spelling of a word might take some time, in which you cannot respond to keystrokes, which could be very annoying to the user. The solution for this should be obvious, namely starting an additional thread for the spell checker. This has a number of nice advantages, namely the logic for the character input thread gets less

[15] or some swirly figures, the latest movie hit, or scantily clad men or women. Whatever preferences.

complicated, and it doesn't have to wait for the spell checking to complete. The operating system will take care of the character input and spell checker thread taking turns.

multi-threading

Running multiple threads, or multi-threading, comes at a cost however. If two or more threads at the same time access the same piece of memory, this can have undesirable or even disastrous results. Take the word processor example again: if the spell-checker thread is checking a word while the user is modifying it, the spell checker is probably checking the wrong word for correctness. And if that word is deleted, chances are you get a dreaded 'blue screen of death', or its equivalent on a non-windows system. To prevent this kind of drama, there is 'thread synchronisation'. Thread synchronisation is all about making sure one thread can get exclusive access to pieces of memory. It if often difficult to determine where and how to do thread synchronisation, which makes multi-threaded programming in potential much harder than normal, single threaded programming.

Interesting enough, the agent computing model provides a possible solution to this. Because agents are autonomous, they can easily send and receive asynchronous messages. Now, if each agent guards his own piece of the memory, each agent can safely modify and read his own piece memory and send messages with results to other agents (provided the agent itself is single threaded, but we'll get to that.) Of course, the message queues themselves still need thread synchronisation, but at least that is highly localised. The bad thing about this is that on a pure message based system, you miss out on rich method naming and type checking. For this reason, we use a 'mixed' approach in our prototype.

6.3.2 Agent threading models

For the discussion above, we can now derive four different options for running several agents on a single computer:

- Single Process, single threaded.
- Single Process, thread pool,
- Single Process, multi-threaded.
- Multi-process.

master process

The autonomy the agent generally increases with each option. The first three have a single process that controls the agents. We will call this the *master process*.

Each option is accompanied with a UML class diagram and a code example that shows the basic differences in terms of agent and master process. Each code example includes an agent with the earlier mentioned 'beat' method that contains a cycle of the agents work. To simulate the autonomy of the agent, this method has to be called continuously.

Single process, single threaded
The option with the least autonomy. This might be your only option if the OS you are running on does not allow multiple threads or processes. It is also the preferred option if the agents have to be synchronised in time, as is often the case in simulation or animation. If you implement some fancy priority and time scheme, agents can have more control over when the are run, and your are edging towards option 3. An extra advantage of this approach is that there is no need for thread synchronisation, since there only ever is one thread.

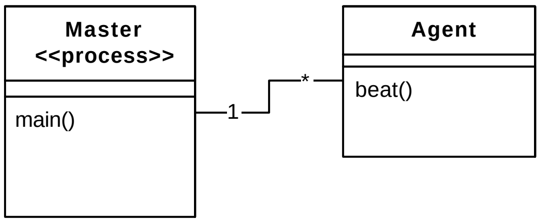


Figure 55: Single process, single threaded.

Fairly simple, One master, many agents, the main method of the master (a process) driving the agents. The master process runs in a continuous loop. Only one agent is shown, although in reality there would be many, each activated in turn.

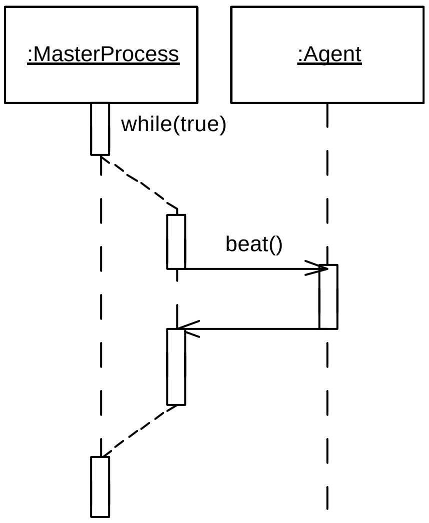


Figure 56: Master process activating the agent.

```
class Agent
{
   …
   public abstract void beat();
}

class Master
{
      private Agent[] _Agents;

      public static void main(string args)
      {
         …
         while (!_Stop)
         {
            for (int counter = 0;counter <
   Agents.length;counter++)
            {
             _Agents[counter].beat();
            }
         }
      }
   }
```

Single process, thread pool

thread pool

A little more freedom. The master process still keeps a tight lid on the threads the agents are allocated, but now it runs each 'beat' in a different thread. This gives an agent a little more freedom, because it can takes as long as it wants during its 'beat', knowing that doing so will still allow other agents to run as well.

Threads are a scarce and thus precious resource on an operating system, so this is a good option if you need to run a lot of agents on the same system, as it allows multiple agents to share the same thread. Application servers often employ a mechanism similar to this, both to save time creating threads, and to avoid having too many threads at once.

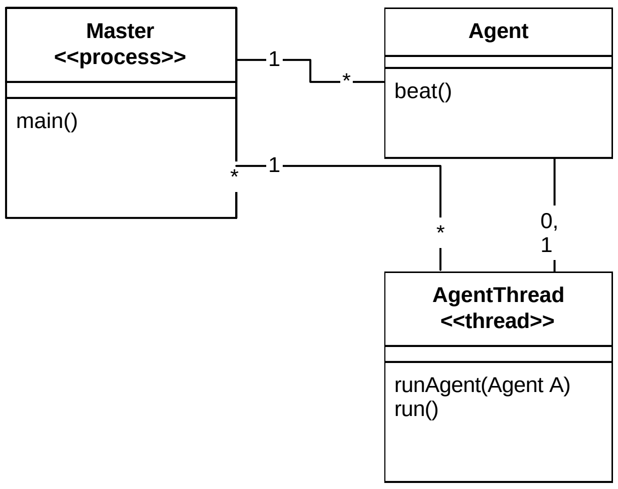


Figure 57: Single process, thread pool.

This option adds a set of 'AgentThread's to the design. As shown in the next sequence diagram, the main process continuously keeps the 'AgentThread's and thus the agents active, but the run() method of the Thread only activates the agent once, not in a cycle.

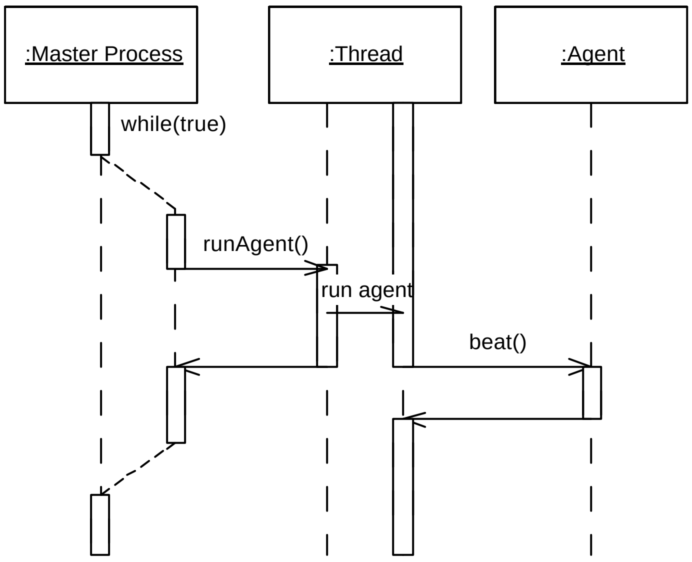


Figure 58: Master process activates threads in a thread pool.

```
class Agent
{
   …
   public abstract void beat();
}

class AgentThread extends Thread
{
      protected boolean _free;
      protected Agent _Agent;
   …
   public runAgent(Agent A)
   {
        _Agent = Agent;
      _free = false;
   }
   public boolean isFree()
   {
      return _free;
   }
      public void run()
   {
        while (!_stop)
      {
         if (!_free)
         {
          _Agent.beat();
               _free = true;
         }
         else
         {
          sleep(…);
         }
      }
   }
}

class Master
{
   private Agent[] _Agents;
      private AgentThread[] _Threads;

   public static void main(string args)
   {
      …
      while (!_Stop)
      {
         for (int counter = 0;counter <
Agents.length;counter++)
         {
          for (int counter2 = 0;counter2 <
_Threads.length;counter2++)
          {
                 if (_Threads[counter2].isFree())
                 {
                        _Threads[counter2].
runAgent(_Agents[counter]);
                 }
         }
      }
   }
}
```

Single process, multi-threaded
This is where you give an agent its own thread. You can choose whether you let the agent start its own thread, or if the master process creates the thread and then hands it over to the agent. In any case, the agent is now owner of its own thread, which it can run or sleep as it sees fit, and it can rely on the underlying operating system to distribute processing time fairly among different agents. It's good to note that having the agent start its own thread is very convenient if the master process is not really dedicated to supporting agents, but an agent just happens to be part of the whole. Having it start its own thread makes to object part of the master process, but otherwise fully autonomous. Note that the agents are still in the same memory space, so they can easily talk to each other.

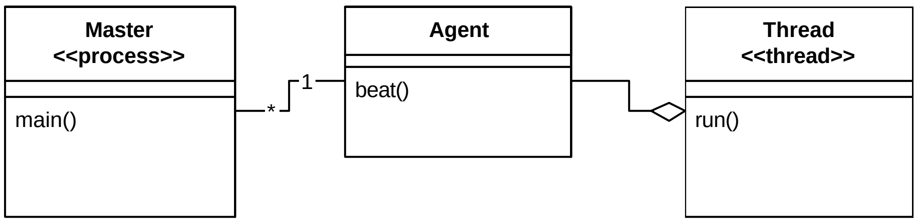


Figure 59: Single process, each agent its own thread.

We see here that the thread becomes part of the agent. The run () method of the thread keeps the agent 'beating'. What the main() method of the master does is not that relevant anymore.

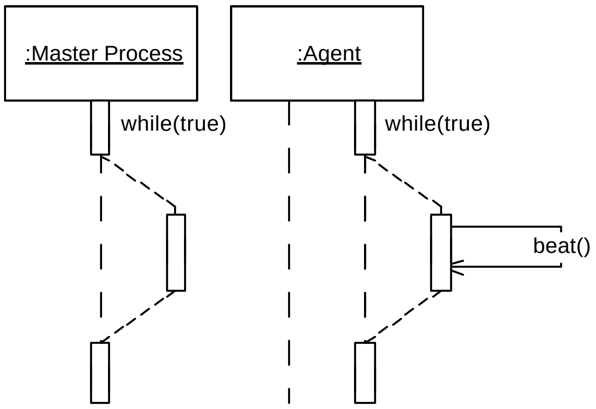


Figure 60: Master process and agent run independently.

```
class Agent implements Runnable
{
   public Agent()
   {
      new Thread(this).start();
   }
   …
   public abstract void beat();

      public void run()
   {
      while (!_Stop)
      {
         beat();
      }
   }
}

class Master
{
   private Agent[] _Agents;
   public static void main(string args)
```

```
    {
       // create agents here
       …
    }
 }
```

Multi process
Finally, you can decide to make each agent a stand-alone application. This might be an option when the agents really live and die with the machine, or when each agents has to be implemented in totally different software / programming language. The agent is a normal application, so it can do anything it just so pleases. However, it is more difficult for two agents to talk to each other, as they have to interact using some from of inter-process or inter-machine communication.

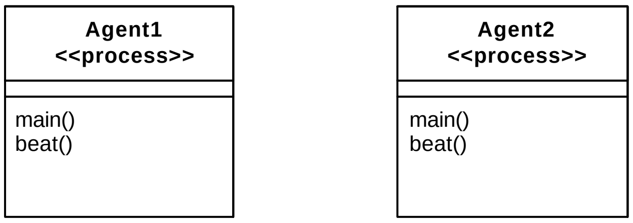


Figure 61: Agents as separate processes.

This looks simple enough. We show two agents for clarity. There does not have to be a relation between the two.

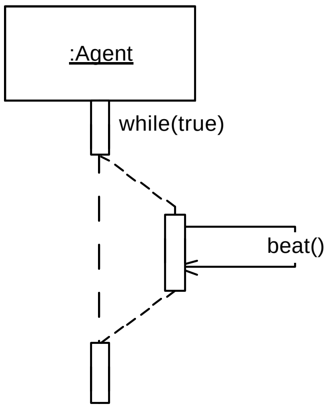


Figure 62: Agent as a process.

The code is for a single agent is also pretty simple:.

```
 class Agent
 {
    …
    public abstract void beat();

    public static void main(string args)
    {
        while (!_stop)
        {
           beat();
        }
     }
  }
```

6.4 Creating mobility

To get mobile agents, two problems need to be solved. First, two (or more) agent spaces should be able to communicate, in order to find out about each other and send the mobile agents back and forth. The second problem is to realise the mobility of the agent, to find a way to recreate the agent in another agent space. The first is relatively easy, but the latter definitely not.

To get two agent spaces to talk to each other, many of the distributed computing paradigms / technologies described in 5.3 will do the trick. You could use TCP / IP and a self-created protocol for communication, and, on a small network, IP-multicast (which is like TCP, but broadcast to many computers at once) to make the agent spaces find each other. Less complicated to program, but requiring additional technology, is to use a distributed object technology like RMI or CORBA to make each agent space a distributed object. An additional advantage would be that the look-up services provided by those technologies could be used to locate other agent spaces.

A big choice to make is whether the mobile agent should be scripted, or if the agent should be programmed in a general-purpose programming language, generally the same as the one used to program the agent space.

The first option means the agent space essentially becomes an interpreter for the agent language. Some instructions in that language would cause the agent space to stop interpreting the agent code and forward the agent in its current state to another agent space. This approach has the clear advantage that you can fully control what the agent can and cannot do. The disadvantages are the scripting language is yet another programming language, and it is very difficult to anticipate everything a programmer might want to do in the agent. However, there are existing implementations that use this approach, most notably Safe-Tcl [10] and TeleScript [37].

If we choose a general programming language as the language for the agent, the problems become more or less reversed. We have to find out how we can make sure the agent can run in another agent space, and what the programmer should refrain from doing to maintain agent mobility.

To find out what it takes to create a mobile agent, we view the agent as an animate object. An animate object has state, behaviour and autonomy (see 4.10.1), and each of those aspects should be recreated in another process, the other agent space.

6.4.1 Moving state

The first hurdle is to recreate an agent's state on another machine. If both the current and the other machine are of the same type and run exactly the same software, this could be fairly easy. The current state of the agent could be copied from memory and send across the wire. However, if that state includes references to other memory locations, that memory should be copied as well, and we quickly run into the same trouble as before. The issue we have here is hardly new. A similar problem occurs if you want to store an object persistently, on a storage medium. Fortunately, the problem has been solved before many times, in various ways.

If the agent is scripted, you need to make sure that each data type you define is translatable in some format that is transmittable over a network and can be recreated in the target agent space. You could define your own encoding scheme, or, rely on the encoding of some distributed computing technology.

If you implement your agent spaces as CORBA objects, you could define the agent state as a structure of arrays of CORBA types. The agent's state variables in the scripting language would map to individual entries in that structure.

For agents written in a general-purpose language, you could use the same trick, but the agent creator has to provide the means to encode and decode the agent state to the transmittable form.

object serialisation

A similar option is to use *object serialisation*. Both (D)COM[16] and Java provide a way to encode an object's state into a stream of bytes and back. If the object holds references to other objects, those are encoded as well. The only catch is that the combined state of all those objects ultimately has to consist of primitive types that can be encoded. This means that arbitrary pointers are off limits, and for instance references to network sockets or local files as well. An additional advantage is that a serialised object can easily be written to disk as well. This is very convenient if you want to shut down an agent space without moving all the agents out first, or to provide fail-over for the agent space.
This option is available both for scripted and non-scripted agents. For scripted agents, it is not really the variable itself that is serialised, but rather the data structure representing that variable in the interpreter.

If we use RMI (Java) to implement the agents and provide inter-agent space communication, the use of serialisation would be automatic. If the agent object is marked as 'serializable', the Java virtual machine will attempt to serialise it. For DCOM it would not be automatic, but it is definitely possible to get the same effect by implementing custom marshalling on the agent object.

[16] COM refers to this as 'structured storage' but the idea is the same.

6.4.2 Moving behaviour

In order to get the same behaviour for the agent, the receiving agent space should of course run the agent using the same code as the originating space. In a controlled environment (say an internal network) we could assume each agent space has the code of all the agents in the network available. Upon receipt of the agent, it could simply recreate the agent object using the readily available code.

However, in a more general case, you would need mobile code (see section 5.3.3). That is, the receiving space should be able to request the agent's code from the originating space. Of course, it should be able to run it as well. For a scripted agent, this is fairly easy, since both spaces interpret the same agent language. For a non-scripted agent, it is slightly more difficult depending on the portability of the code. On a Win32 platform, we could make sure each agent's code resides in its own DLL, and we could enable the agent spaces to transmit those DLLs to each other. For Java, it is still fairly easy because the target machine by definition is able to run the same code as well. Again, RMI will automatically transmit an object's code to the other virtual machine if it doesn't have it yet. For DCOM, it could be made part of the custom marshalling.

6.4.3 Moving autonomy

So far, the agent simply was a mobile object, but the final step is to transport the aspect that is unique to agents: autonomy. Unfortunately, this is also the hardest, at least in its most general form. For a scripted agent, it is possible, by serialising the internal state of the interpreter (as related to the agent) and sending it along with the agents' state (and code). The TeleScript language we mentioned before does just that. For a non-scripted agent, you would want to send the thread running the agent, in its current state, but this is prohibitively difficult, and would require operating system support. Even in Java it is not possible, because each JVM implementation has to implement threads in its own way.

Not all hope is lost however. We think there would rarely be a need to be able to move the agent at every possible moment in its execution. Generally the agent initiates the move itself, and its programmer can take necessary measures to make sure the agent can resume execution based on its object state (so not including any local variables). The 'beat' method concept described in 6.2.1 can help, as we can move the agent in between beats: Once a beat has ended, we can stop the thread, serialise the agent, send it to another space, deserialise it and start a new thread there. The agent wouldn't miss a beat.

6.5 Mobile agent pattern

Based on the idea's of the previous section, we now present a mobile agent design pattern. It defines the roles and responsibilities of the two pieces of the mobile agents paradigm: mobile agents and agent spaces. We follow the same structure as in the standard book on design patterns [43]. Each pattern is described in a number of sections with fixed names:

- **Class** – gives a basic categorisation of the pattern.
- **Motivation** – describes the underlying ideas of the pattern.
- **Applicability** – states in what situation the pattern can be used.
- **Structure** – shows a class diagram with the participating objects.
- **Participants** – gives a textual description of the roles of each class in the pattern.
- **Collaboration –** shows how the participants interact together.
- **Consequences** – describes what the resulting behaviour is of a system using the pattern, and lists possible attention points.
- **Implementation** – shows a code example of the pattern.

And now the description of the mobile agent pattern:

Class
Object behavioural

Motivation
To enable autonomous objects, referred to as agents, to move to another machine and continue their task there. This can be advantageous if there are differences in availability and bandwidth between machines in the network, or to provide the object local access to resources on several machines, which could substantially reduce network traffic. The pattern allows for a very flexible computational architecture where running tasks can be moved around at will.

Applicability
Allows for increased network flexibility and availability. For instance, a continuous task can be moved off a user's machine onto a central server, after which the user can disconnect. If the user reconnects, possibly on the another machine, the task can move to the user again, and interact with him or her. It also provides an alternative for distributed servers: instead of anticipating and exposing all possible uses of the server the server could receive mobile agents, bringing with them their own code to interact with the primitives available on the server. It can be viewed as a generalisation of the query mechanism as employed by database servers. It is also a possible model for massively parallel computing, where the task is initially created on one machine, and subsequently moved to many servers for parallel computation, after which they move back again with the results.

Structure

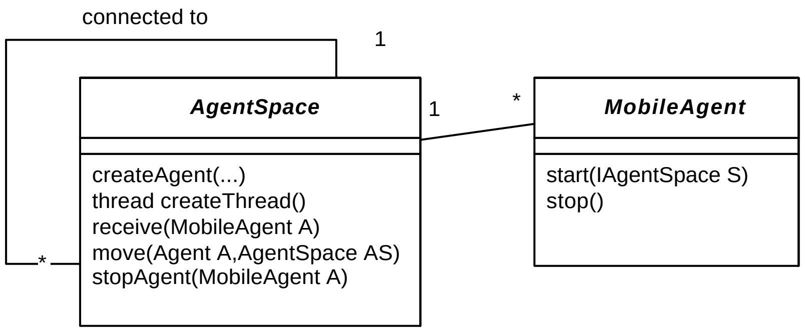


Figure 63: Mobile agent pattern.

Participants
Agent Space: a process that hosts the agents.

Mobile Agent: the autonomous object that can move between different agent spaces.

Collaboration
This pattern supports three basic operations: creating new agents, moving agents and stopping agents. We describe each with a sequence diagram and accompanying text.

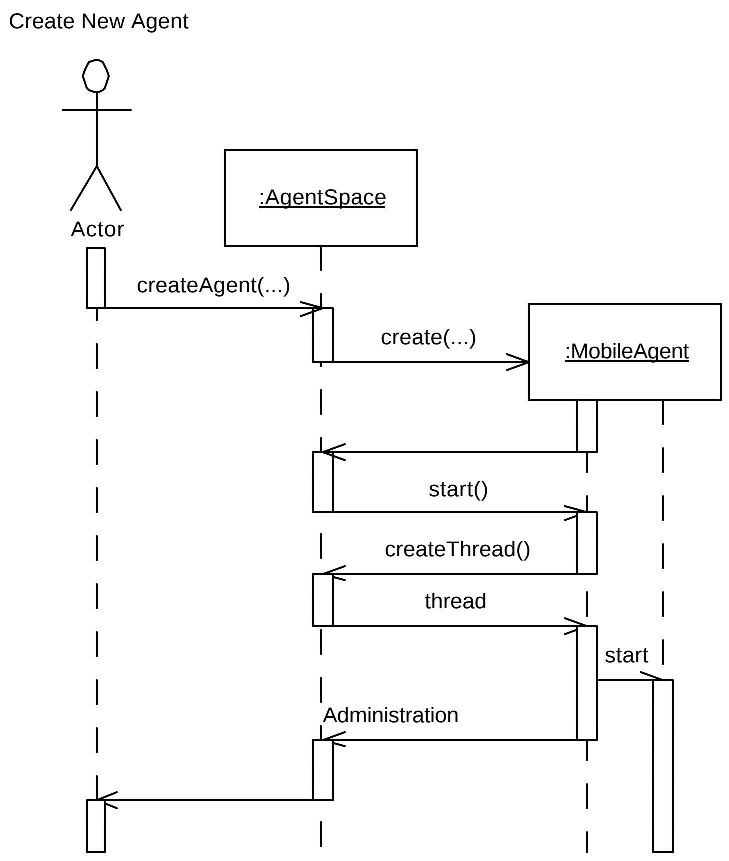


Figure 64: Create a new agent.

What makes this operation non-trivial is the fact that the agent has to be given its own thread, which has to be started. It is not strictly necessary to give the agent its own thread, but then this interaction is replaced with similar operations to register the agent some sort of multitasking mechanism. In addition, we don’t want the mobile agent to start its own thread for security and control reasons. This construct allows the agent space to keep a watchful eye over thread allocation.

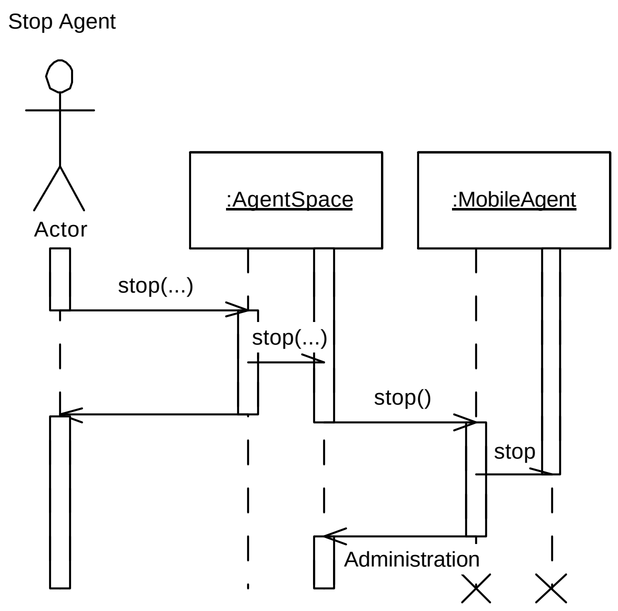


Figure 65: Stop mobile agent.

Again, what is a bit of work is having the agent stop its thread. In addition, we show two activation lines for the agent space, to indicate it needs a separate thread to stop the agent. This is because we want the agent to be able to stop itself, and without a separate thread, this becomes cumbersome to implement. Also it is generally a good idea to have the starting and stopping of agent threads is handled by a separate thread, for the same reasons as described above.

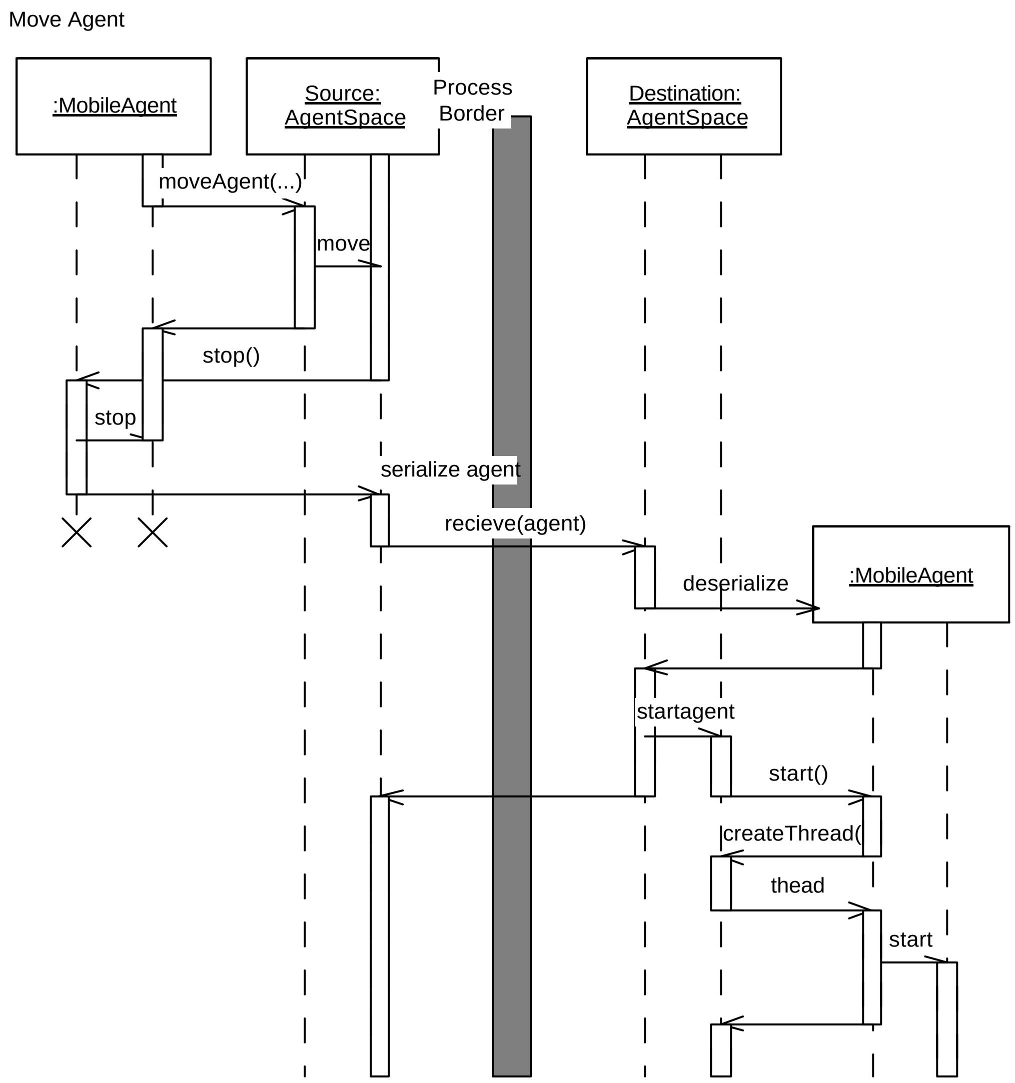


Figure 66: Move mobile agent.

This last operation is largely a concatenation of the previous two: we see the agent is first stopped in the source space, and started in the destination space. The agent itself, being an autonomous object, initiates this operation. The one thing extra in this picture is the ‘receive’ method, but it entails a lot. It requires there is inter-process communication possible between the two agent spaces, and that the agent’s state and behaviour can be transmitted in some form to the other agent space. These two points are not addressed by this pattern, but the previous sections describe several possible ways this can be accomplished.

Consequences
This pattern shows how to simulate the move of an active, autonomous object to another process, something that is generally not supported by

programming languages and / or operating systems. It does mean that the moved object is able to resume execution solely based on its object state, and any data it wishes to take with it should be kept in that state.

Implementation
Implementing this pattern in Java (using RMI) is fairly straightforward, thanks to Java's excellent support for multi-threading, object serialisation and the portability of Java code. However, as discussed above, it should definitely be possible to implement this using other languages / platform, either by creating a scripted agent language or largely simulating the behaviour of Java's RMI.

Special care must be taken with the thread creation and destruction. For instance, Java will not allow a thread handling a remote call (as the thread handling the 'receive' on the destination space) to create a new thread. This means this task should be delegated to a separate thread, as shown in our interactions.

Sample code
We will not put sample code here, but the base classes for the prototype in chapter 7 implement this pattern, so that code should suffice.

6.6 Conclusion

In this chapter, we showed how agents and mobile agents could be implemented. We identified three different ways you can implement agents: scripted, using a framework, or do-it-yourself (DIY).

Scripted
A scripted agent is written in some kind of (currently) proprietary agent language, such as TeleScript. Applications such as Microsoft Exchange also allow for the creation of 'scripted' agents, although those are defined using a GUI. The big advantage of this type of agents is that they are relatively easy to create, since most of the complexities are hidden away in the run-time environment running the agents. The turn side of this method is of course that the agent scripting environment might not offer all the features required, which at best leads to possibly complicated enhancements to the agent scripting environment.

Agent Framework
Another option is to develop agents using some kind of agent software framework, such as IBM's Aglets. The difference between such a framework and an agent scripting environment is that with a framework, the agents are developed in a normal, full-fledged programming language, often the same language as the framework itself is developed in. The big advantage of this approach is that it is a lot easier to add features to the agents that are not offered by the framework itself, thanks to the full power of the programming language.

DIY
Still, it might be that none of the existing frameworks or scripting environments suits your needs, either because they don't have some crucial features you need, or because they don't integrate well with some other technology you might need. The only option left is then to develop the agents yourself, in a general purpose programming language, such as Java or C++. Just because no framework can help, doesn't mean you are on your own completely. We named several types of technologies that we think could help building an agent based application. Most notably:

- object oriented programming languages
- distributed objects
- messaging systems
- application servers

The remainder of this chapter focussed on various implementation strategies for DIY agents.

We first showed how the various components of an agent: the heart, the brain, the memory, the sensors, and the effectors can be modelled in an UML model. In the simplest case we see the brain and the memory as an integral part an agent class, with separate classes for the various effectors and sensors. The heart can be a thread, or some other autonomy mechanism, as described later in the chapter. For intelligent agents, it makes more sense to separate the brain and the memory, and have the brain influence the memory. The agent class then makes sure that the sensor outputs are placed in the memory, that the brain performs the reasoning, and that changes in memory state are translated into effector activations.

Seeing the sensors and effectors as part of the agent is not the only approach however. For larger systems, it might make more sense to separate the sensors and effectors into *services*, which can be used by multiple agents and even by non-agent applications. And another ways is to make a particular kind of sensor or effector be part of one single agent, and have the agents utilise each other to get to various inputs or actions. Such a system is sometimes referred to as a federation of agents.

We listed four basic options for creating multiple autonomous agents in a single system: single process, single-threaded; single process, thread pool; single process; multi-threaded; multi-process.

Single process, single threaded – is simple in the sense that there are no multithreading issues. A single thread activates all the agents, one at the time. The disadvantages is that each agent has to voluntarily give up control after a short amount of time, otherwise the other agents can't do anything in the mean time.

Single process, thread-pool – allows each agent to a lot more independently, while still limiting the amount of threads in the system. This is very good approach, but a thread-pool is non-trivial to implement.

Single process, multi-threaded – means you simply give each agent its own thread. This is relatively simple, and you in effect entirely rely on the underlying operating system to make sure each agent gets a chance to run.

Multi process – means that you see each agent as an independent application. This makes sense if the agent for instance has to come up or down with the computer itself, or if there are several agents on the same machine that are implemented using a different technology. A disadvantage is that it is a lot more cumbersome to enable the different agents to communicate.

Next, we showed how to create mobile agents. This entails three things, we have to move the state of the agent (its data), the behaviour of the agent (its code), and the autonomy.

Moving state means the data the agent maintains has to be transferable to another machine, so encoded and decoded in some kind of standard data format. Distributed object technology provides convenient ways of doing this, a process usually referred to as *serialisation*.

Moving behaviour implies mobile code, code that in some way can be executed on the machine receiving the agent. Section 5.3.3 discusses this topic in detail. The java platform form a particular good example of this, since java bytecode can run on any platform for which there is a Java Virtual Machine (JVM).

Moving autonomy is the toughest of the three. It means moving a running thread to another machine, something that is only supported in some very high-end, special purpose operating systems. This means we have to 'trick' this feature in some way. We do this by taking advantage of the 'beat' method concept, and have the agent only move after a single beat terminates. The one disadvantage this has is that it is not completely transparent to the agent programmer that the agent is moved. He or she has to save any state he or she wants to maintain in the agent's state, and write the 'beat' method in such a way that it can resume its task for the beginning of the 'beat' method. Still, we believe this is a small price to pay for having autonomous agents.

Finally, we presented the mobile agents design pattern, which consolidates both the use and implementation of the mobile agents in the standard design pattern format.

CHAPTER 7 Mobile user agents framework

With all the foundations firmly laid, we are now ready to show how everything fits together, by designing and building a prototype mobile agents application. Our application consists of two main parts: *Octarine*: a generic framework for supporting mobile user agents, and *WebWatcher*: A web front-end to the framework together with a set of 'Internet service' agents. We choose this application over the other options mentioned in chapter 5 because it is not that difficult to build, and the mobility of the agents is very visible. When designing this application, we followed the basic methodology of chapter 4, although note that while the actual design went through several cycles, we only present the final results here.

7.1 Application description

Here we describe the basic functionality of the prototype application, divided in to three parts, the Octarine framework, the WebWatcher application and the WebWatcher agents.

7.1.1 Octarine

Octarine

Octarine is a framework that hosts agents that perform routine tasks for a user, with those agents able to jump back and forth between the user's machine and a central server. We called it Octarine after the eighth colour, the colour of magic[17] [23], because we like to think the moving of the agents is a bit magical. An interesting thing is the agents are neither just client, nor server, but both. Each agent can be thought of as a little application, that dependent on what it has to do next, runs on either the client or the server.

This picture shows the basic context:

[17] Octarine is described as a sort of greenish yellowish purple, and can only be seen by wizards and witches.

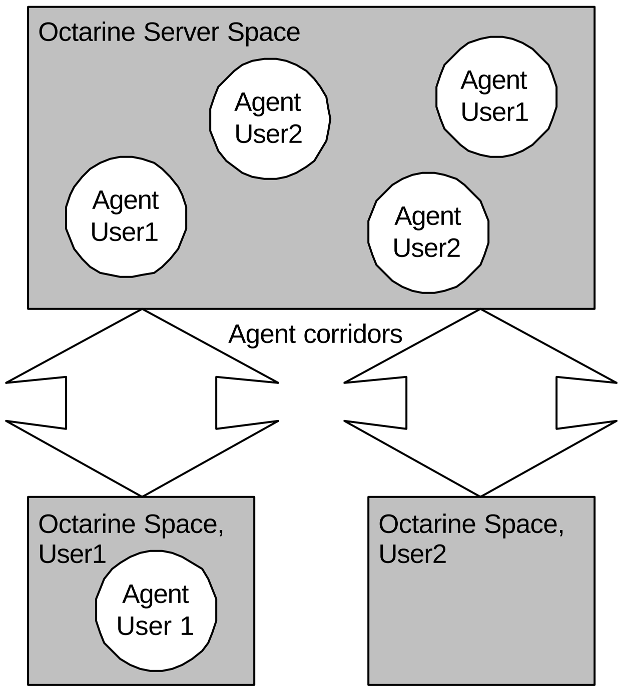


Figure 67: Mobile user agents.

server space, user space

It is just a slight refinement of the mobile agent picture in 5.4. We now recognise a *server space* and *user spaces*, and we have labelled each agent according to its owner. The server space obviously runs on a central server, and the user spaces run on the client machines. The user can connect and disconnect this space from the central server as he or she sees fit. An agent belonging to a certain user will never move to the user space of another user, at least not in the current setting. We of course want to support an arbitrary number of users, but we show just two for simplicity.

7.1.2 WebWatcher

GUI

The WebWatcher application is a web front-end to the framework, tailored for Internet enabled agents, and enabling the agents to interact with a user using a Graphical User Interface (GUI). To accomplish this, WebWatcher includes a user space that is enhanced to provide these facilities.

Because WebWatcher is web-based, it allows a user to access his agents from anywhere he or she pleases, without first having to install software. Using the interface, the user can create new agents, view the status of his or her existing agents, edit an agent's parameters, and stop an agent. While the user's overview page is open, the user's agents can contact the user if they want to.

Usability of the application would increase further if the user could optionally install an application that runs on his or her machine permanently. This would allow the agents to alert the user at all times, however, that requires too much additional effort at this time.

7.1.3 The agents

Now, the agents. The initial design of this application includes just two types of agents, the WebWatcher agent, which shares its name with the application itself, and the FileFreak agent. Both of these agents are Internet enabled agents, they need an Internet connection to operate.

WebWatcher

smart bookmark

The WebWatcher agent is a kind of *smart bookmark*. The user gives it an URL to watch, and the WebWatcher will alert the user if the text on the page has changed. This saves the user tremendous effort in revisiting a page over and over again, and finding there is nothing new to report. For this prototype, this is all we will do, but there are plenty of possibilities to enhance this, by making the alert depend on a specific text appearing, have it include image changes, or have it monitor a whole tree of pages.

FileFreak

FileFreak is a download assistant. Instead of the having the user wait ages while a download trickles through at far less than the connection speed, the FileFreak will download the file, and deliver it to the user full speed. The user can disconnect as soon as the agent is installed, and will be alerted as soon as the download is complete. Of course, for this to work, the bandwidth between the agent space server and the user has to be sufficient, otherwise nothing is gained.

These are just two agents this application can support, but there are many others: We can make a tailored agent that watches auction sites for items the user wants, or presents stock quotes in a neat little package. We can have an agent searching e-commerce sites for the best price of a product, implement the over-used travel agent, etc.

7.2 Technology selection

It should be pretty obvious what technology we use to implement this application. We already argued in **Error! Reference source not found.** that Java (using RMI) has many nice features that make it fairly easy to implement mobile agents, so Java is our pick for the Octarine framework. This also means the WebWatcher user space is build as a Java applet. Keeping with the Java theme, we use Java Servlets to respond to and generate the HTML pages for WebWatcher.

Now a context diagram, showing the basic software components, how they relate to each other, and using what technology:

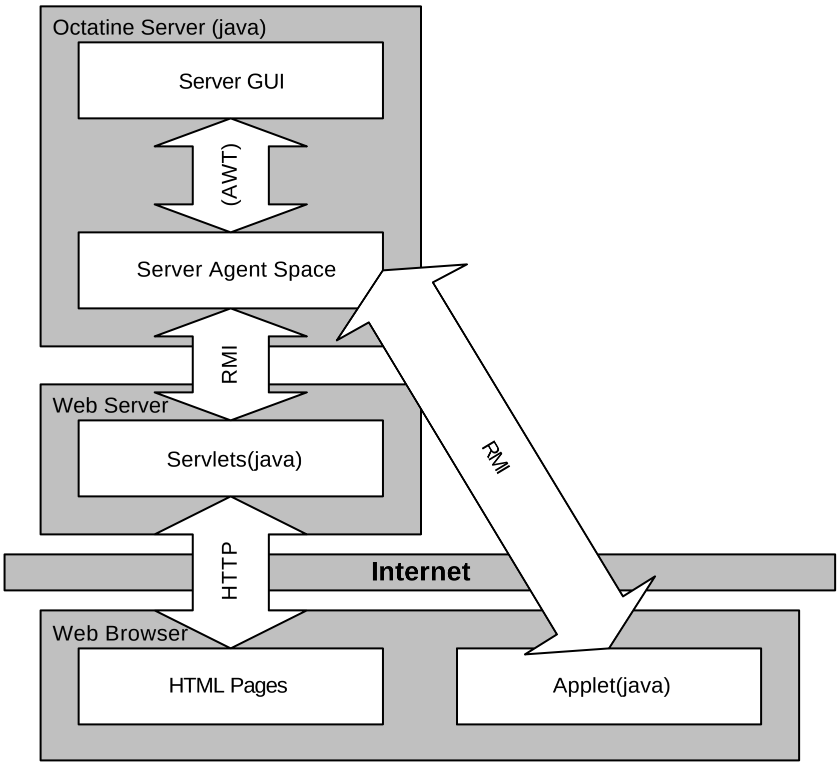


Figure 68: Context diagram.

The grey boxes show the basic software components, the Agent Space server, the web server, and the web browser. The white boxes show what sub-components live inside those components (note that the agents are omitted from this picture). The text in the arrows shows which protocols/technologies are used for communication between those components.

The Web Server and Agent space server need to reside on the same physical machine, because an applet is only allowed to contact the server from which it originated. However, for a production system, there are some (router) tricks to work around this limitation.

7.3 Use cases

The previous sections described the basic functionality of the application. Now we do so more thoroughly, by describing use cases for each of the possible interactions with the system. We use the same division in three components again, but in a slightly different order, which makes more sense for a user's point of view.

7.3.1 WebWatcher application

The WebWatcher application consists of six use cases, which mostly deal with various agent administration tasks.

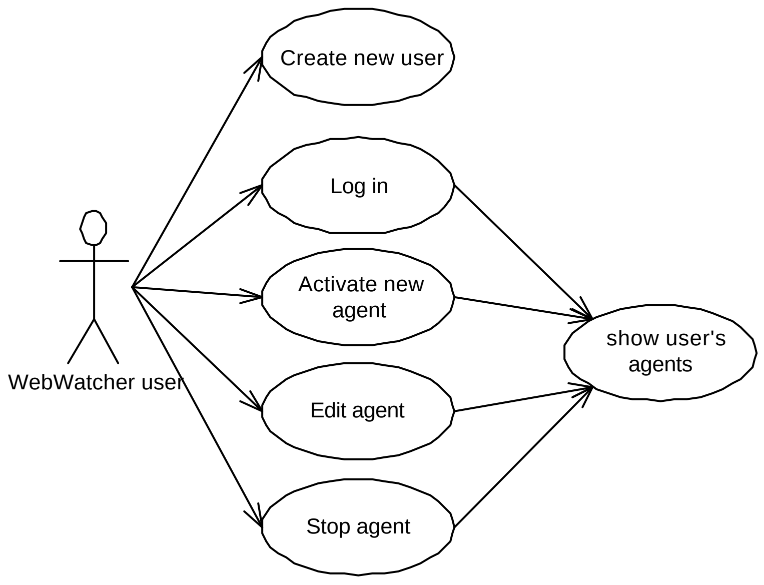


Figure 69: Interface use cases.

Create new user
User selects to become an application user
Site presents a user name and a double password prompt
User types in username and the password twice.
Site verifies that the user name doesn't exist, that the passwords match, and creates a new user entry.
The user is returned to the home page.

Log in

```
User access the site
Site presents password prompt
User fills in username and password.
IF (the username and password are correct)
   Site grants access.
   INCLUDE (Show user's agents)
ELSE
   Site presents an error screen.
   The user is returned to the home page.
```

Activate new agent
User selects to activate a new agent
Site presents a list of possible agents classes
User selects desired agent
INCLUDE (Show user's agents)

Edit agent
Site presents a list of the user's agents.
User selects agent to change
Site sends edit request to Server Space for agent.
INCLUDE (Show user's agents)

Stop agent
Site presents a list of the user's agents.
User selects agent to stop
Site asks to confirm the stop
User confirms the stop.
Site stops the agent.
INCLUDE (Show user's agents)

Show user's agents
The site presents a screen of all the user's agents. This screen also includes the applet hosting the agents. From this screen, the user can exercise the various agent administration tasks and of course interact with his or her agents.

7.3.2 Agents

Here we describe the two types of agents the user can create. The WebWatcher agent that can watch a web pages for a user and report changes, and the FileFreak agent that can assist in downloading large files on congested networks.

WebWatcher agent

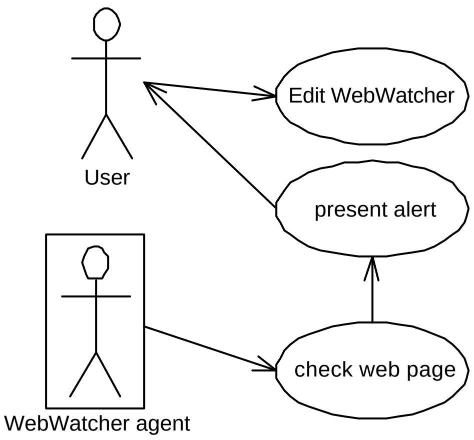


Figure 70: Web watcher use cases.

Note: Although 'present alert' is only included in 'check web page', it is presented separate because it has an additional beneficiary, the user.

Edit web watcher
User selects to edit the WebWatcher (Or creates a new WebWatcher)
Web watcher moves to the user's space.
Web watcher presents a dialog with the URL it is watching
The user changes URL to watch.

Web watcher moves back to the server.

Check web page

```
WebWatcher requests the web page it is watching
Web server returns web page
WebWatcher compares web page to old web page
IF page is different
    INCLUDE (present alert)
```

Present alert

```
WebWatcher shows a dialog with the URL to the (now changed) page to user.
IF (The user wants to follow the link)
    User clicks link.
ELSE
   User dismisses dialog.
```

FileFreak Agent

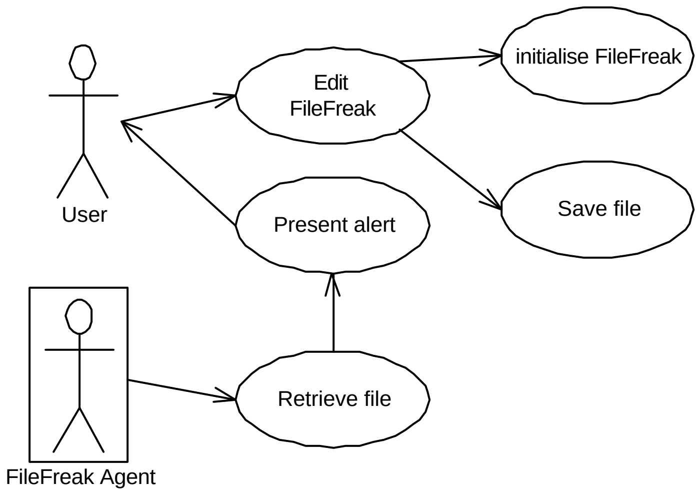


Figure 71: FileFreak agent use cases.

Edit FileFreak agent

```
User selects to edit the FileFreak (Or creates a new FileFreak)
FileFreak moves to the user's space.
IF (a file is retrieved)
   INCLUDE (Save File)
ELSE
   INCLUDE(Initialise FileFreak).
```

Initialise FileFreak

FileFreak presents a dialog with the URL to the file to download.
User changes the URL.
FileFreak moves back to the server.

Save File

FileFreak agent displays save file dialog.

User directs the FileFreak to a folder where the file should be placed.
FileFreak agent saves file to disk.
FileFreak agent terminates.

Retrieve file.
FileFreak Agent starts FTP or HTTP session with server that holds the file.
FileFreak Agent gets file.
INCLUDE(Present alert)

Present alert
FileFreak agent moves to user's space
FileFreak present a dialog with a message that the file is now ready for retrieval.
IF the user wants the file now.
  INCLUDE (Edit FileFreak)
ELSE
  User dismisses dialog.

7.3.3 Octarine Server

The use cases for the Octarine server can be divided into four categories: Those initiated by other agent spaces, by the agents, by remote clients of the server, and by a server administrator. We discuss hem in that order. A few use cases are marked with a + sign. This is to indicate that the initial prototype will not have this functionality in this way. They have been included nonetheless, because we figured they describe how the system should work.

Other spaces

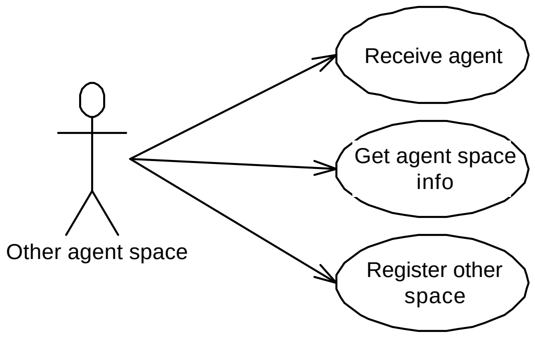


Figure 72: Octarine server, other spaces use cases.

Receive agent
The other agent space sends an agent to this agent space
Server receives the agent, reconstructs it and confirms the reception.
Server activates the agent.

Get agent space info
Other agent space requests info on this space.
Server returns info uniquely identifying this space.

Register other space
Other agent space registers itself with this space.
Server asks info on other space and adds it to the list of connected spaces.

Agents

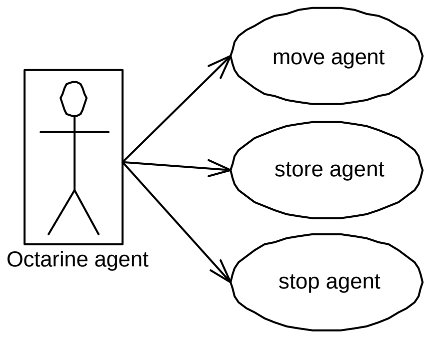


Figure 73: Octarine server, agent use cases.

Move agent
The agent asks the current agent space to move it to another agent space
When that space is available, the agent space stops agent and sends it to the destination agent space
Destination agent space receives the agent and starts it.

Store Agent
Agents asks to be stored in persistent storage
Server stores agent to persistent storage.

Stop agent
Agent asks to be stopped.
Server stops the agent.

Remote Clients

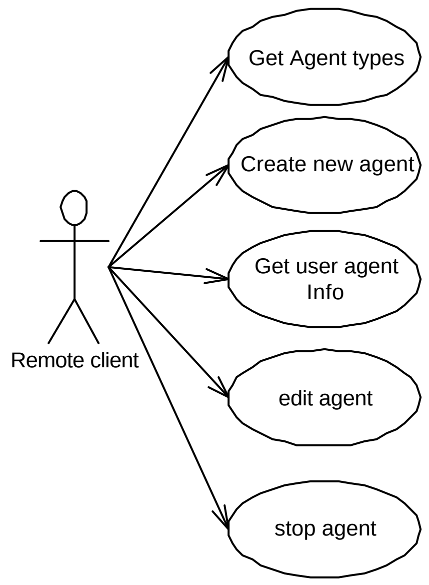


Figure 74: Octarine server, remote client use cases.

Get agent types
Remote client asks available agent types.
Server returns a list of the available agent types

Create new agent
Remote client asks to create a new agent
Server creates the new agent.

Get user agent info
Remote client asks a list of a user's agents
Server returns a list of that info on those user's agents

Edit agent
Remote client asks to edit an agent
Server sends edit request to agent.

Stop agent
Remote client asks to stop a particular agent
Server stops that agent.

Administrator
Note: The use cases whose names start with a plus are included for completeness, but are not implemented in the prototype.

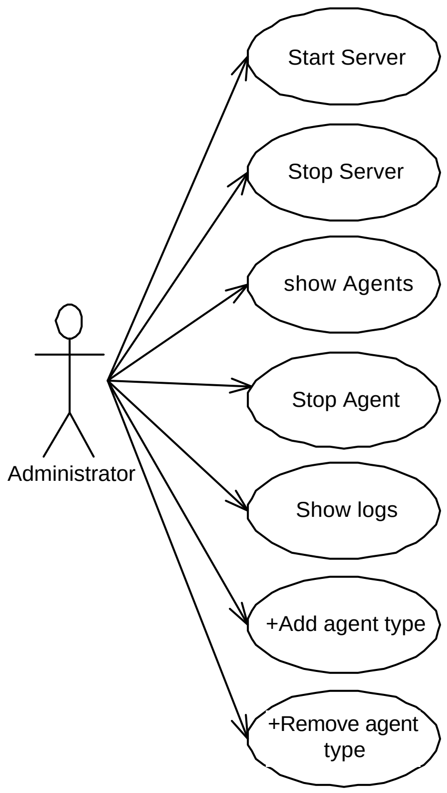


Figure 75: Octarine server, administrator use cases.

Start server
Administrator starts server
Server starts,
Server loading all agents from persistent storage
Server registers itself with RMI registry.

Stop server
Administrator asks server to stop.
Server removes itself from the RMI registry.
The server shuts down, stores all agents, refuses new requests, and stops.

Show agents
Administrator asks for a list of all agents
The server presents a list of all the agents in the system.

Stop agent
Administrator selects agent to stop
Server asks to confirm the stop
Administrator confirms the stop.

Show logs
Administrator selects to view logs
The server presents a screen with log entries.

+Add agent type
Administrator selects to add new agent type
The server presents an open dialog.
The Administrator selects the .jar files to add.
The server integrates the new agent type in to the agent base.

+Remove agent
Administrator selects to remove an agent type.
The server presents a list of all the present agent type.
Administrator selects agent class to remove.
Server asks to confirm the removal, warning if there are outstanding agents of this type and if there are connected user spaces with this type of agent in them.
Administrator confirms the removal.
Server removes all instances of this agent type, and removes the agent from its agent base.

7.4 Class Diagrams

Now we present the class diagrams for the Octarine server and WebWatcher application. We start bottom up, from the Octarine base classes, and build up from there, ending with the web interface. The class diagrams all went through roughly three iterations: We first created an initial class definition based on the use cases. Then we developed object interaction diagrams for all the use cases, which let to the addition of most of the methods on the objects. A final run led to some refinement and streamlining of the object relations and methods.

7.4.1 Octarine base classes

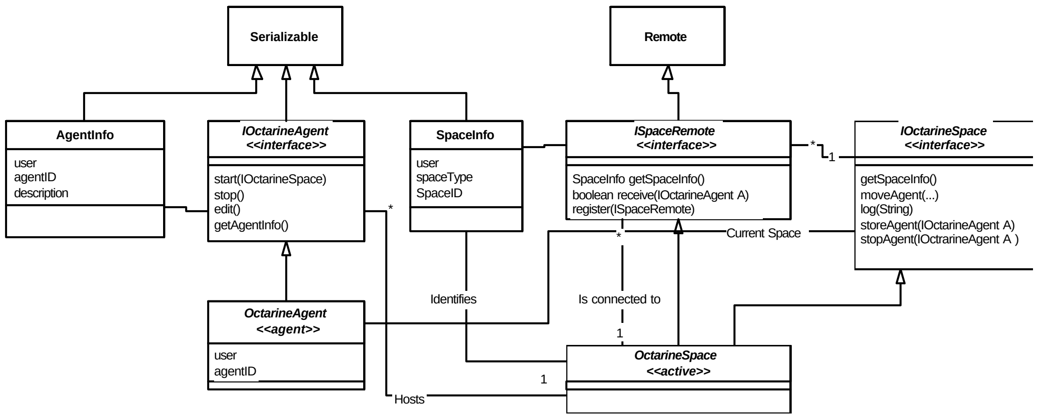


Figure 76: Octarine base classes.

These classes are essentially an instance of the mobile agent pattern (6.5), but with some added flavour to make it all work. The design is centred on the three interfaces 'IOctarineAgent', 'IOctarineSpace' and 'ISpaceRemote'.

The agent space and the agents only know about each other based on these interfaces, which makes sure no class specific dependencies are introduced. We have separated the remote interface ‘ISpaceRemote’ from the ‘IOctarineSpace’, because only those methods should be exposed when the Octarine space is accessed remotely.

‘OctarineAgent’ is an abstract base class for Octarine agents. It implements the start and stop methods, but leaves the rest to the derived class. We use the ‘single process, multithreaded’ autonomy model from 6.3.2, so each agent has its own, dedicated thread.

‘OctarineSpace’ is an abstract base class for Octarine spaces. It implements the functionality for receive agents, the stopping of agents, moving of agents.

The two info classes: ‘AgentInfo’ and ‘SpaceInfo’, are just simple informational structures that uniquely identify agents and agent spaces, respectively. They serve as a simple ‘persistent reference’ implementation. They are ‘serializable’, so they can be sent to another machine.

7.4.2 Octarine Server classes

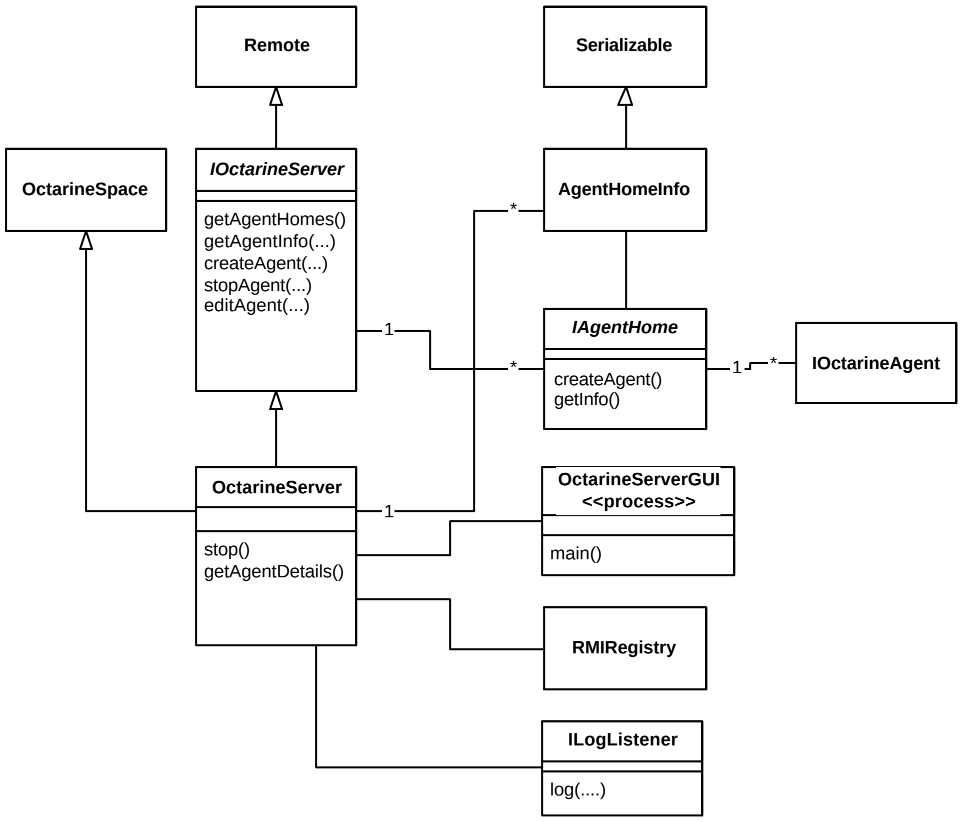


Figure 77: Octarine server

The core of this picture is the ‘OctarineServer’ class, which implements the remaining functionality of the ‘IOctarineSpace’ interface, as well as the ‘IOctarineServer’ interface, which holds the methods exposed to remote clients of this server. It delegates logging functionality using a Listener interface, which in our case is implemented in a GUI class (not shown).

The ‘IAgentHome’ interface is an application of the factory pattern. We use a home interface (and implementing object) to construct new agents. In this way, the server only has to know about a set of home interfaces, and doesn’t care about any specific types of agents. The ‘home’ naming is inspired by a similar construct in Sun’s Enterprise Java Beans [46]. The ‘AgentHomeInfo’ class is again a simple information class uniquely identifying a certain home interface.

We won’t say too much about the ‘OcatrineServerGUI’, since this thesis is not about building Java GUIs. It suffices to say it enables administrative use cases, and consists of a few more classes than shown here. It is a process, and its main tasks are to interact with the administrator and keep the GUI updated.

7.4.3 Octarine Applet classes

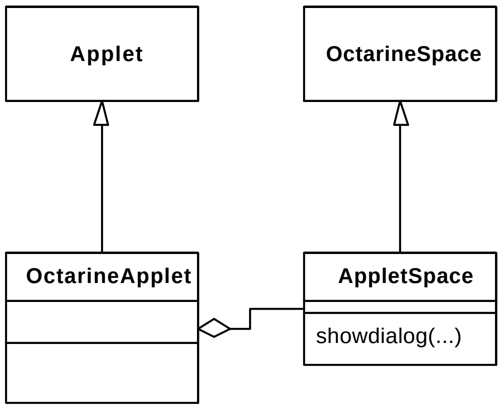


Figure 78: Octarine applet.

Of course, the applet is fairly simple. We've tried to keep it as lean as possible, because this component has to be downloaded to the user. The applet is composed of a very simple applet implementation and a simple 'AppletSpace', derived from the 'OctarineSpace' base class. What is interesting, is that it implements the 'showDialog' method, used by the agents to communicate with the user.

The Applet uses the RMI registry to locate the server and register itself with it.

7.4.4 Servlets

servlet

We now show the *servlets* that make up the WebWatcher front-end. Servlets is a Java technology that allows you to extend the functionality of some web servers. A servlet is mapped to the name of a certain folder on the web server. Normally, when the folder (or a file in the folder) is requested by a client, the web server would return that folder, or the file in that folder. If there is a servlet mapped to the folder however, the web server will instead activate the servlet and pass the requested URL to it. The servlet is expected to return a stream of bytes, which is send to the client by the web server.

The application also includes quite a few classes that facilitate the generation of dynamic screens, but those have not been included here.

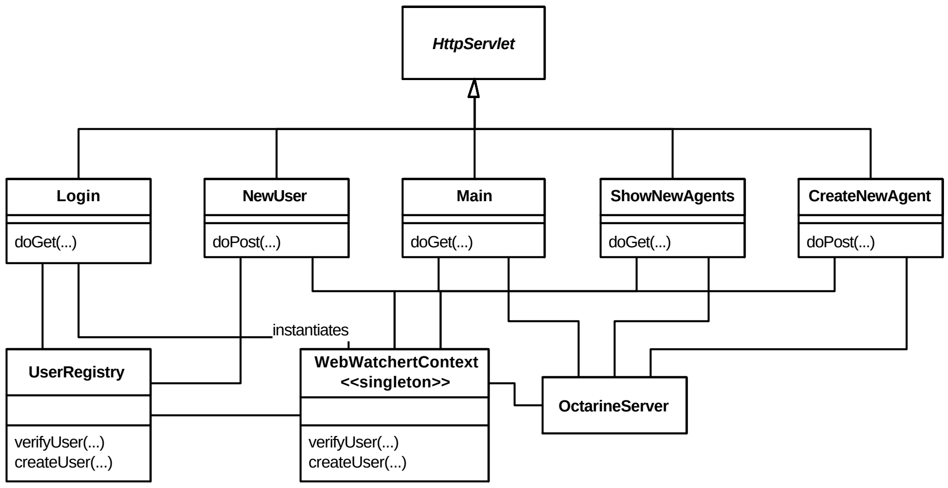


Figure 79: WebWatcher servlets.

The WebWatcher front-end consists of four servlets, and two helper classes.

The ‘Login’ servlet authenticates a user and allows access to a user’s personal agents. The ‘NewUser’ servlet creates a new user in the application. ‘Main’ generates the main screen, listing a user’s agents, and also making sure the ‘OctarineApplet’ on the main screen knows which user’s agents it should accept. ‘ShowNewAgents’ queries the server for the available agent types, and displays a selection list on the screen. ‘CreateNewAgent’ creates a new agent for a particular user.

The ‘UserRegistry’ is a simple property-file based user database, keeping usernames and passwords used by the application. The Login servlet instantiates a ‘WebWatcherContext’, which represents the common memory of the WebWatcher application. All other servlets can access this same context object. This construct allows us to avoid duplicating logic in all the servlets. The ‘WebWatcherContext’ for instance keeps a reference to a ‘UserRegistry’ object and to the ‘OctarineServer’.

The following picture shows the relations between the screens and the servlets of the WebWatcher application.

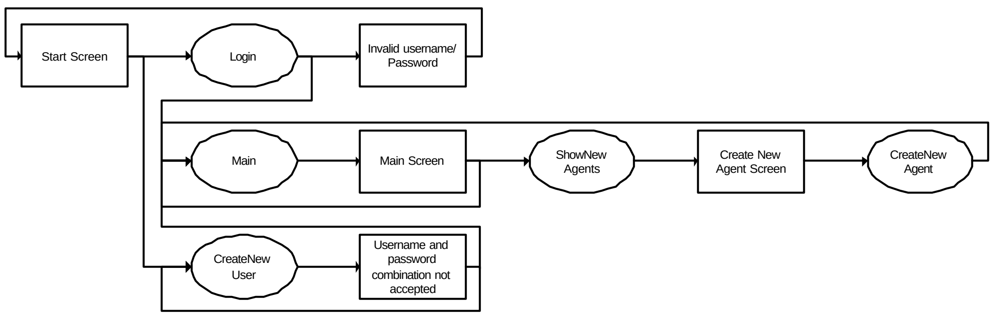


Figure 80: WebWatcher page flow.

Here we see that the basic flow of the application is rather simple. From the start screen, you can either login, or register as a new user. Once logged in, you interact with the application through the main screen, which 'loops' into itself, with one exception, the creation of a new agent shows a screen with a list of the available agents.

7.5 Points of interest

The complete prototype entails quite a few sequence diagrams and a few hundred lines of code. We could of course list and discuss all of those here, but this document is big enough as it is. We thought it would be a better idea if we picked a few interesting sections of code and show how the mobile agents concept works out in practice.

One thing we did not introduce before was the inclusion of a use case oval in a sequence diagram. It corresponds to an 'INCLUDE' statement in a use case description. It means that at that point, interaction continues on the sub use case.

7.5.1 Main screen

Most of the action of the application happens on the main screen. The main screen shows the user's agents, and contains the 'OctarineApplet'. The sequence diagram for this use case, 'Show user's agents' is as follows:

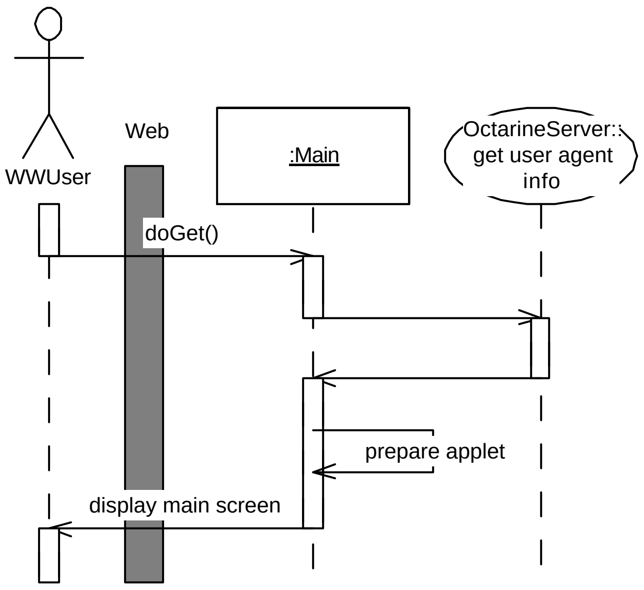


Figure 81: Display of main screen.

The code for this looks like this:

```
   class Main extends servlet
   {
      public void doGet(HttpServletRequest
   request,HttpServletResponse response) throws
   IOException
      {
         WebWatcherContext Context =
   WebWatcherContext.getContext();
         HttpSession S = request.getSession(false);
         if (S == null)
         {
         Context._Engine.display(new ErrorScreen("No
session"),response);
         return;
         }
         try
         {
            String UserName =
   (String)S.getValue(USERNAME);
            Iterator Agents =
   Context._Server.getAgentInfo(UserName).iterator();
            Context._Engine.display(new
  MainScreen(Context._MainScreenFile,new
  StringScreen(UserName),new
  AgentDataScreen(Context._AgentDataFile,Context._Agent
  RowFile,Context._SeparatorFile,Agents)),response);
         }
         catch (RemoteException E)
         {
            Context._Engine.display(new
   ErrorScreen("Unable to reach Octarine Server.
   please try again later."),response);
         }
      }
   …
   }
```

The first thing the servlet does it getting a reference to the 'WebWatcherContext', to get access to all the application specific data. Throughout this and code you see references to 'Context._Engine' and various 'Screen' classes. These are classes to facilitate the display of dynamic[18] html pages. This design was created before we familiarised ourselves with Java Server Pages (JSP). If we would have known JSP at the time, we would have used that instead.

We see the servlet first checking if there is a session, and if not, display an error page. It then proceeds to get the user name from the HttpSession, and retrieving the user's agents from the Octarine server.
The user name, which is also passed to the applet is inserted into the 'MainScreen', as well as an 'AgentDataScreen' containing the user's agents.

7.5.2 Agent creation

As the use case shows, the creation of an agent happens in two stages, first the user is presented with a screen describing all the agents types. The user selects one, and that an agent of that type is subsequently created. This reflected in the sequence diagram for this use case.

[18] This is dynamic as in containing information specific to the current user, not in the HTML 4.0 sense of dynamic.

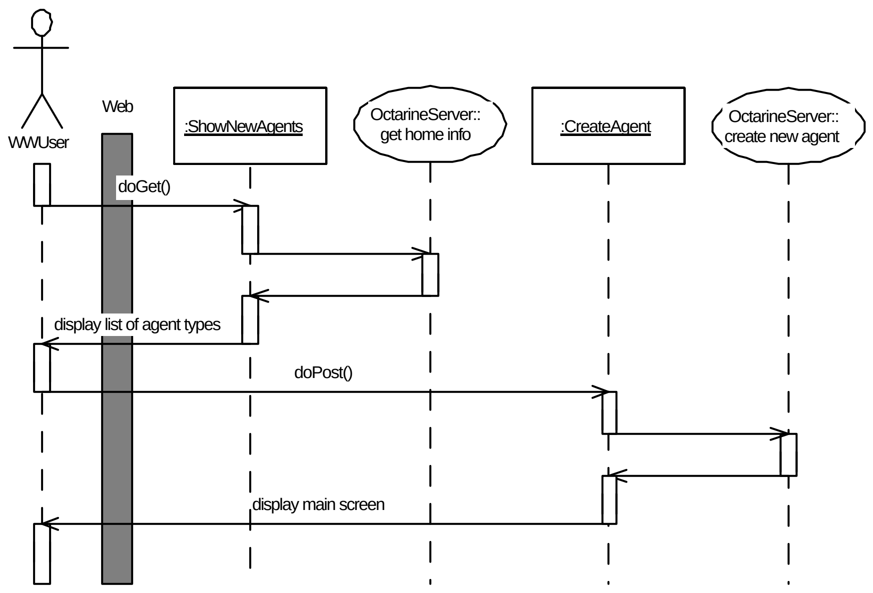

WWUser
Web
:ShowNewAgents
OctarineServer:: get home info
:CreateAgent
OctarineServer:: create new agent
doGet()
display list of agent types
doPost()
display main screen

Figure 82: Servlets: create agent.

So far, nothing special. Two servlets, one responding to the screen listing the different agents available, and one for creating the agent. The really interesting stuff happens in the two server use cases. The functionality in ‘get home info’ is a lot like ‘get agent info’, which we’ve already shown when discussing the main screen. Let’s look into ‘create new agent.’

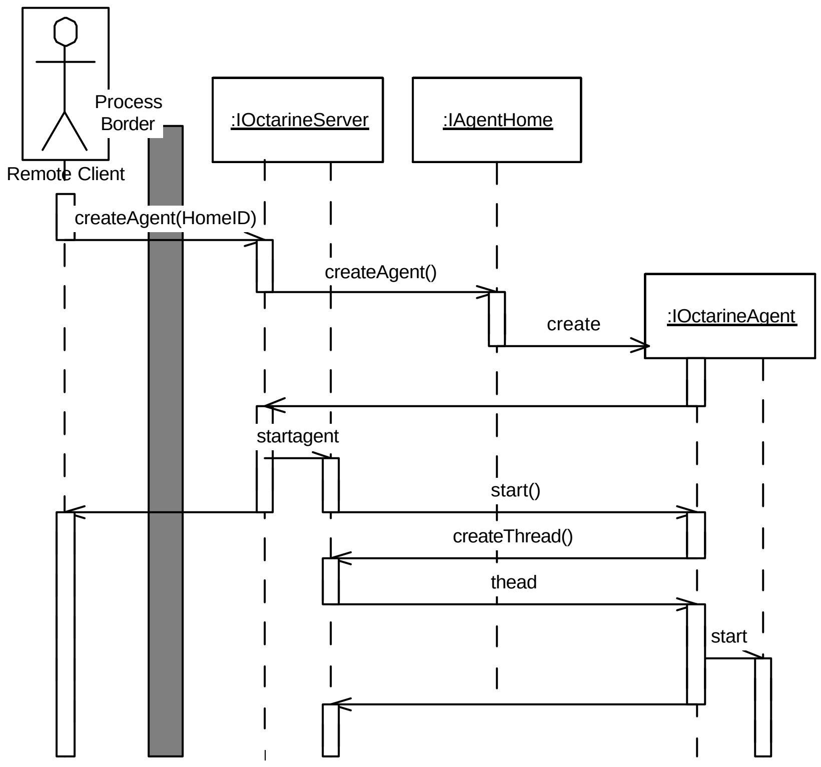


Figure 83: Server: Create new agent.

There are two things to note about this: first, the use of ‘IAgentHome’ as a factory for the agent, and second the actual starting of the agent by a separate thread of the ‘OctarineSpace’ (the base class). The consequence of this is that while the agent is created, it might not yet be started when control returns to the servlets. That is something we need to take into account there.

In code, this is what some of it looks like:

First in the RemoteServer: (The piece of the Octarine Server implementing the remote interface.)

```
public class RemoteServer extends
java.rmi.server.UnicastRemoteObject implements
IremoteServer
{
   public boolean createAgent(java.lang.String
HomeID,String User)
   {
      IAgentHome Home =
(IAgentHome)_Server._AgentHomes.get(HomeID);
      if (Home == null)
      {
         _Server.logWarning("received creation
request for unknown agent home ID " + HomeID +".");
         return false;
      }
      IOctarineAgent Agent =
Home.createAgent(generateID(HomeID),User);
      if (Agent == null)
      {
         _Server.logError("AgentHome did not create
an agent");
         return false;
      }
      if (!_Server.receive(Agent))
      {
         _Server.logError("Server failed to accept
new agent");
      }
      return true;
   }
}
```

We see the RemoteServer first creating a new agent using the agent's home interface, and then making '_Server' (the actual OctarineServer class) 'receive' the agent. This is the same method that is used to send an existing agent to the Octarine Server. From the point of view of the server, there is no difference between a newly created agent and an existing one.

The really interesting stuff happens in the 'OctarineServer' (or actually its base class):

```
public abstract class OctarineSpace extends
UnicastRemoteObject implements IOctarineSpace,
IRemoteSpace, Runnable
{
   public boolean receive(IOctarineAgent A)
   {
      logInfo("Agent received for User: '" +
A.getUser() + "'");
      synchronized (_Incoming)
      {
         _Incoming.add(A);
         try
         {
          _Incoming.wait();
         }
         catch (InterruptedException E)
         {
          logWarning("Wait intterupted.");
         }
      }
      logInfo("Agent started for User: '" +
A.getUser() + "'");
      return true;
```

```
    }
  …
  }
```

The ‘OctarineSpace’ adds the agent to a list of incoming agents, and then waits on the incoming list until the ‘OctarineSpace’’s main thread actually activates the agent. This way, a client can be sure the agent is really running once the method call ends.

The actual start happens in the ‘OctarineSpace’’s main thread:

```
public void run()
{
   …
   synchronized(_Incoming)
   {
      Iterator I = _Incoming.iterator();
      while (I.hasNext())
      {
         IOctarineAgent A = (IOctarineAgent)I.next();
         _Agents.put(A.getAgentInfo()._AgentID,A);
         A.start(this);
         logInfo("Agent started.");
      }
      _Incoming.clear();
      _Incoming.notifyAll();
   }
…
}
```

The ‘OctarineSpace’ adds runs through the list of newly added agents, and then adds the agent to its own internal list of running agents. The agent is then started, the incoming list is emptied, and all the waiting threads are notified.

7.5.3 Editing an agent

Responding to the edit request in the servlet and on the server is relatively straightforward, so we won’t show it here. It does get interesting in the server. Each agent implementation can be slightly different, but sequence diagram shows how it’s done in the WebWatcher agent, which should be fairly typical:

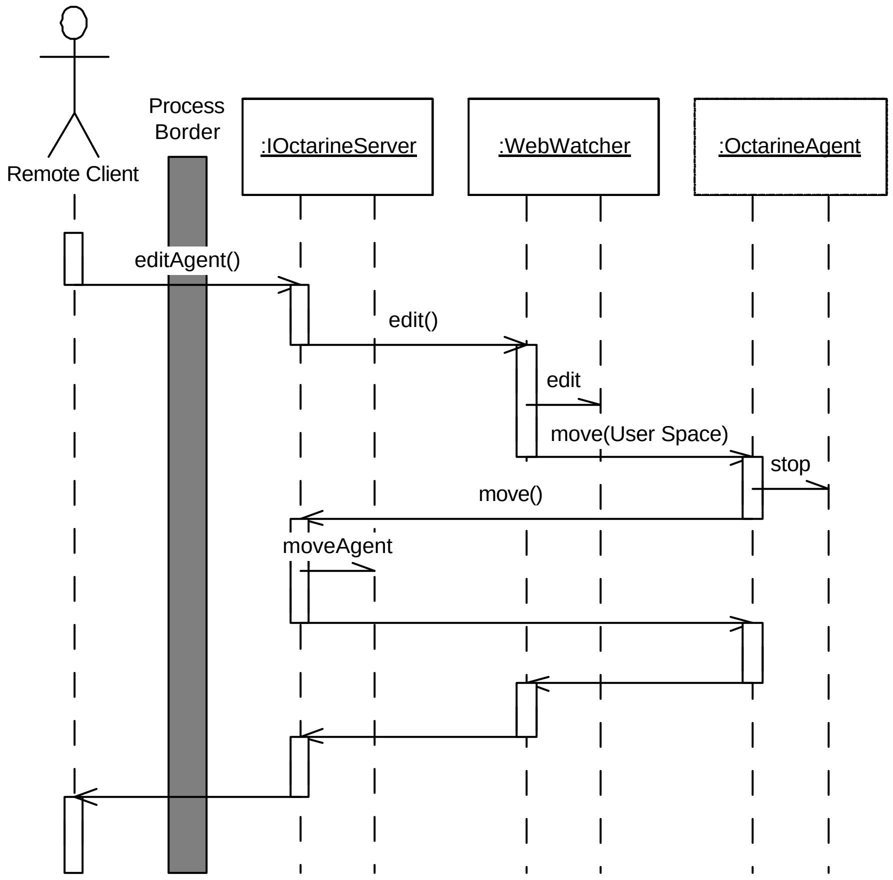


Figure 84: Edit agent (WebWatcher).

You see that as a result of the edit request, three asynchronous messages get send: the 'edit' the agent sends itself, a 'stop' as side-effect to the move, and a 'move agent' as a result of the move request. Here's how that works out in code:

```
public boolean edit()
{
   _Edit = true;
   move(new SpaceInfo(SpaceInfo.ST_USER,_User));
   return true;
}
```

Two things happen: the flag '_Edit' is set, which will cause the agent to pop up and edit dialog upon arrival, a 'move' is executed, and the 'SpaceInfo' constructed in this way means 'A user space that has that the user of this agent associated with it.'

Here's the 'move' method, in the base class, 'OctarineAgent':

```
public void move(SpaceInfo SI)
```

```
{
   _Stop = true;
   _Space.moveTo(SI,this);
}
```

The ‘_Stop’ flag is set, and the ‘OctarineSpace’ hosting this agent is asked to move the agent to the other space. Because the ‘_Stop’ flag is now true, the agent will stop running after the current beat finishes. This means an agent also issues the ‘move’ itself, and knows that on the next ‘beat’ it will be in the new space.

‘moveTo’ in ‘OctarineSpace’:

```
public boolean moveTo(SpaceInfo SI, IOctarineAgent
A)
{
   synchronized (_Outgoing)
   {
      Iterator I = _Outgoing.iterator();
      while (I.hasNext())
      {
         if (((OutgoingEntry)I.next())._Agent == A)
         {
          return false;
         }
      }
      _Outgoing.add(new OutgoingEntry(SI,A));
   }
   return true;
}
```

The first bit is just to make sure the agent did not already issue a move request. The core is the ‘_OutGoing.add’ call that adds an entry to the ‘Outgoing agents’ queue.

What happens to those three messages? Well, each of those gets picked up by another thread, the ‘stop’ message cause the agent’s thread to stop, as shown in the code example in 6.3.2. The addition of the ‘move agent’ has the following effect: (In ‘OctarineSpace:run()’, object thread)

```
synchronized(_Outgoing)
{
    synchronized(_Spaces)
    {
        Iterator I = _Outgoing.iterator();
        while (I.hasNext())
        {
                OutgoingEntry O =
(OutgoingEntry)I.next();
                Iterator I2 = _Spaces.iterator();
                while (I2.hasNext())
                {
                      ConnectedSpace CS =
(ConnectedSpace)I2.next();
                      boolean Move = false;
        //decide if this is the space to move to..
        …
        // decision made.
        if (Move)
        {
                      try
                      {
                      O._Agent.stop();
                             if
(CS._Space.receive(O._Agent))
                             {
                             I.remove();

   _Agents.remove(O._Agent);
                             }
                      }
                      catch (RemoteException E)
                      {
                      logError(E.getMessage());
                      }
        }
        }
     }
   }
```

It works roughly as follows: for each entry in the queue, the list of connected spaces is searched for a space that matches the space the agent has to move to. (The matching logic is omitted because made this code segment rather long.). If that space is found (move is true), the agent is stopped, and send to that space. Then this space is cleaned up, by removing the agent from the queue and from the current agent list.
At this moment, the agent has arrived at its current destination, the user space. After the agent is activated (as shown in Figure 66), the agents 'beat' method of WebWatcher gets a chance to run again:

```
protected void beat()
{
…
   if (_Edit)
   {
      if (_Space.getSpaceInfo()._User.equals(_User))
      {
         WWPanel P = new WWPanel(_URL);
         if (_Space.showDialog(P))
         {
          _URL = P.getURL();
         }
         move(new
SpaceInfo(SpaceInfo.ST_SERVER,null));
      }
```

```
          }
     …
     }
```

If the '_Edit' flag is set, the agent first checks if it indeed is in its owner's space, and if yes, construct a new panel with the URL it is currently watching. It then asks the user space to display a dialog to the user. If the user confirmed the change, the current URL is updated. After showing the dialog, the agent moves back to the server space.

7.5.4 WebWatcher at work

The following diagram sequence diagram shows the WebWatcher at work. If it needs to notify the user, it displays a dialog to the user. Its core operation is the retrieval of a web page, and moving to the user space if that page has changed.

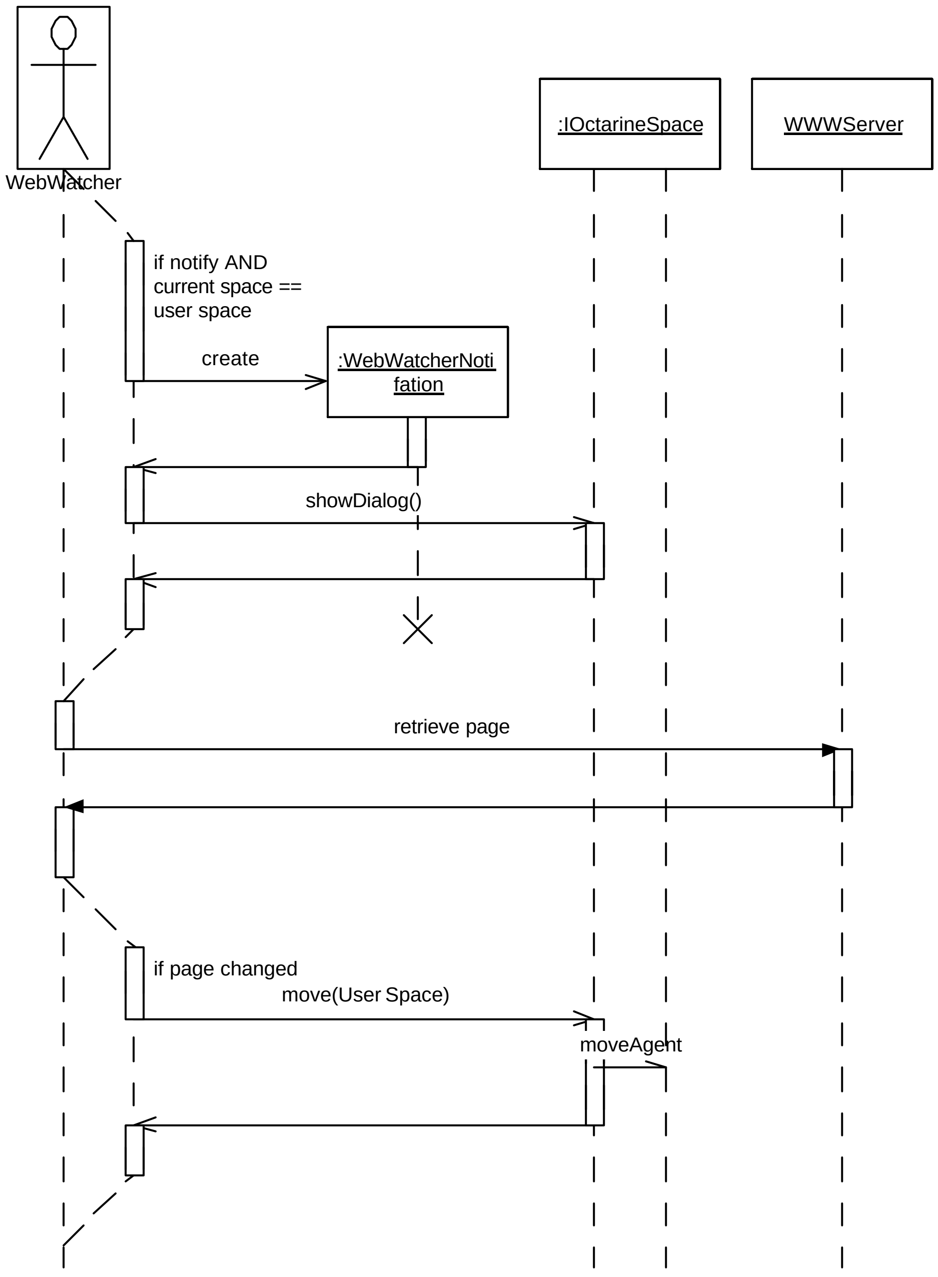


Figure 85: WebWatcher at work.

Now the equivalent piece in the WebWatcher’s ‘run’ method:

```
public class WebWatcher extends OctarineAgent
{
   public void run()
   {
      …
         if (_Notify)
         {
            if
   (_User.equals(_Space.getSpaceInfo()._User))
            {
                    WebWatcherNotification P = new
   WebWatcherNotification(_URL);
            _Space.showDialog(P);
            _Notify = false;
            move(new
   SpaceInfo(SpaceInfo.ST_SERVER,null));
            }

         }
         try
         {
            long Value = parse();
            if (Value != _CheckSum)
            {
                    _Notify = true;
            }
         }
         catch (Exception E)
         {
            …
         }
```

The first should look familiar, it is nearly identical to the code to edit the agent : if the agent is in its users space, it shows a dialog to the user, clears the flag that shows triggered the display of the dialog, and moves back to the server space. Note that both the 'edit' and 'notify' code are examples of effectors. You can think of the panels (dialog windows) as the actual effectors if you will.

The second bit is the core brain of the agent. It makes a call to its one sensor, the 'parse' method, and compares the result of it with the previously recorded value. If it is different, it set's the '_Notify' flag, which in time triggers the mechanism shown above. The actual code in the parse method is quite complicated, and involves setting up sockets, initialising HTML parsers and a few other interesting bits. Its implementation is left as an exercise to the reader.[19]

7.5.5 Applet initialisation

The most interesting stuff happens in the 'init' method of the applet. This is where it sets up its user space and registers itself with the Octarine Server:

[19] There ! We've always longed for an opportunity to say that !

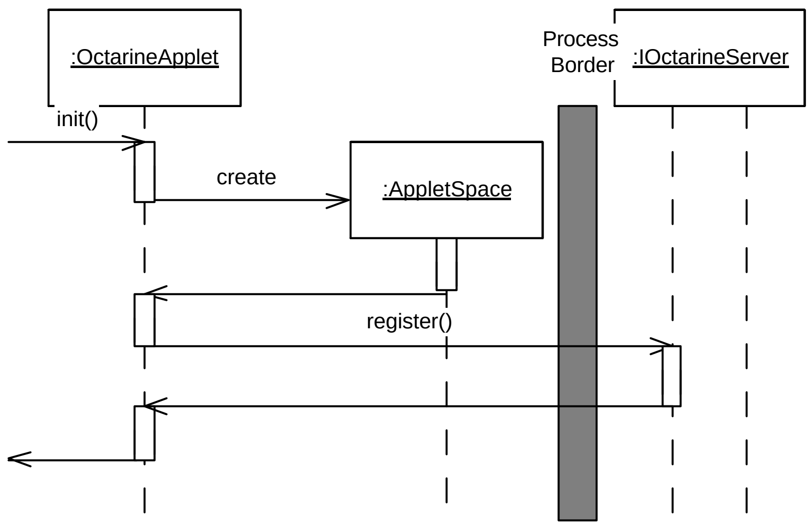


Figure 86: Applet initialisation.

And in code:

```
public class OctarineApplet extends Applet
{
…
   public void init()
   {
      try
      {
         Component C = getParent();
         while (!(C instanceof Frame))
         {
          C = C.getParent();
         }
         _Server =
(IRemoteSpace)Naming.lookup("www.webwatcher.com/Oct
arine/OctarineServer");

         _Space = new
AppletSpace(getParameter(USER),(Frame)C,_Server,_Se
rver.getSpaceInfo());

         _Server.register(_Space);

   System.out.println(_Server.getSpaceInfo()._Type);
      }
      catch (Exception E)
      {
         …
      }
}
```

We first see the applet finding the parent frame it is in, since the user space needs a frame to display dialog windows. It then looks up the server using a standard RMI call. The user space is initialised with the user ID by getting it from the applets' initialisation parameters. Finally, the user space is registered with the Octarine server. From this moment on, the applet might receive agents from the server.

7.5.6 Server start-up

The last piece we want to show is the start up of the Octarine server application. An administrator start the application, which gets the whole thing rolling:

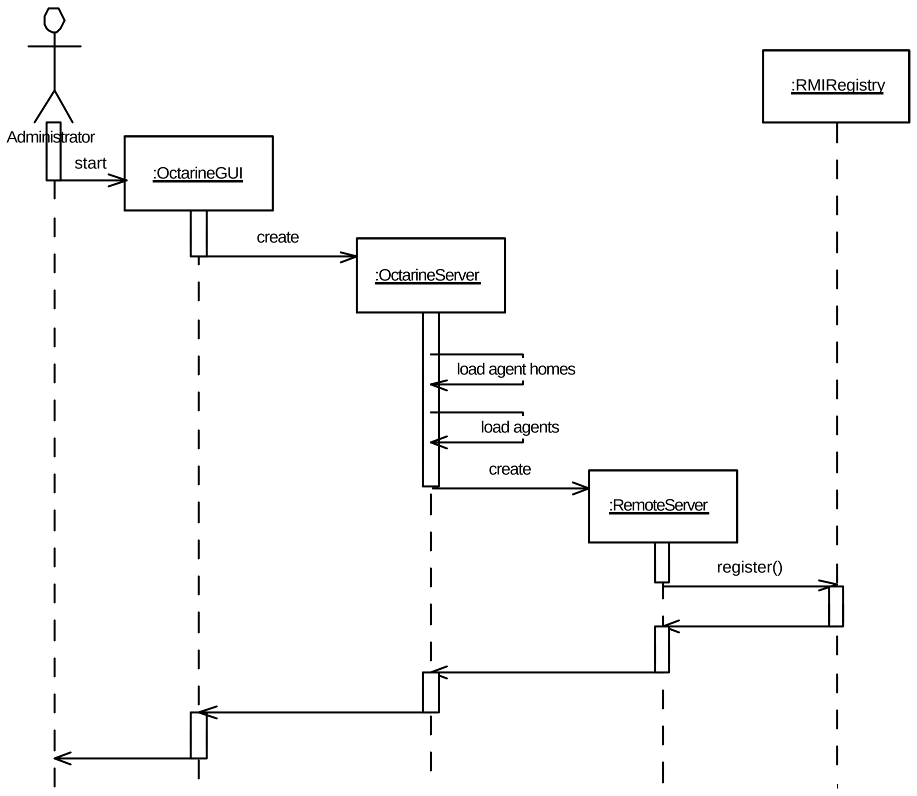


Figure 87: OctarineServer start-up

This is start-up expands over the constructors of several objects, for which the code is shown below:

```
public class OctarineGUI extends Frame
{
…
   public OctarineGUI(String title)
   {
      try
      {
         _OctarineServer = new
OctarineServer(_LogView);
      }
      catch (java.rmi.RemoteException E)
      {
      …
      }
   }
}
```

This shows the construction of the actual server object in the constructor of the GUI. The _LogView parameter passed in is a callback that is used to show log messages in a log message window.

```
class OctarineServer extends OctarineSpace
{
   public OctarineServer(ILogListener L) throws
RemoteException, java.net.UnknownHostException
   {
      super(SpaceInfo.ST_SERVER,null);
      _LogListener = L;
      logInfo("Octarine Server Initialising");

      …
      loadAgentHomes();
      loadAgents();

      try
      {

   Naming.rebind("/Octarine/OctarineServer",this);
      }
      catch  (java.net.MalformedURLException E)
      {
         …
      }
      logInfo("Initalizing remote server");

      _RemoteServer = new RemoteServer(this);

      …
      logInfo("Octarine Server Initialisation
done.");
   }
   …
}
```

This shows the constructor of the ‘OctarineServer’. The first thing that happens is the initialisation of the base class. Note that we pass in the type of space, in this case ST_SERVER. The base class is generic, and doesn’t know whether it is a server space or not. The next thing that happens is the loading of the different agent homes available, and the loading of current agents from persistent storage. Finally, the ‘RemoteServer’ is initialised, which makes this server visible to the outside world.

7.6 Conclusion

In this chapter, we showed the design and implementation of a mobile agents based application. This application consists of two parts: Octarine, a generic mobile user agents framework, and WebWatcher, a Web front-end build on the framework, supporting two types of agents. The first agent, also called WebWatcher, works as a kind of smart bookmark, informing the user when a certain web page has changed. The second is FileFreak, which can take care of downloading a large file on a congested network, and then forward it to the user full-speed.

7.6.1 Design process

For the design of the application, we roughly followed the methodology as outlined in chapter 4. We started with a high-level description of the functionality and requirements of the application. From this, we derived which technologies we were going to use to build the application. In this case, we chose Java/RMI for the Octarine framework, and servlets and an applet for the WebWatcher front-end.

We then detailed the functionality of the application in a set of use cases. These use cases included use cases for user’s using the agents and agents using the Octarine server and whatever other resources they needed. The use cases were used to do a first pass of developing the class diagrams for the application.

For the Octarine framework, we additionally used the mobile agent design pattern found in chapter 6. As we defined sequence diagrams for all use cases, we made sure that the use cases could be implemented using the classes in the object diagram. As we built the sequence diagram, we modified the class diagram to add or modify methods on the classes that were needed by the sequence diagram.

At this point, we had enough information to start the implementation of the application. We highlighted some of the more interesting pieces of code:

We showed how the main WebWatcher screen, the screen displaying all a user’s agents, is build by querying the Octarine server and building and a list of the user’s agents.

An agent is created by selecting an agent type and creating a new instance of that type of agent. The new agent then by itself decides to move to the user’s machine and interact with the user directly.

A running WebWatcher agent uses an HTTP socket to retrieve a web page (only the HTML) from a web server. It then compares a digest of the web page with a digest made earlier. If the digests are not identical, it asks the server to be moved to the user’s machine. If the user connects to the server, it will transport the agent to the user’s machine.

At start-up, the server has to load and activate a set of persisted agents from disk. Only when they are all activated the server announces its availability to other machines.

7.6.2 Reflection on the methodology

During the design process, we clearly saw the advantages of the additional modelling constructs we introduced in chapter 4.

Without allowing agents as actors, describing how the agents interacted with the server part and other parts would have been awkward. We would have to include those descriptions in other use cases, were they would have been out of place.

The sequence diagrams without the double lifelines would have been a lot less clear, since activations originating from different threads would have mingled, and there would have been ‘magic’ activations all over the place.

7.6.3 The future of WebWatcher

WebWatcher is at its current state still very much a prototype application. Still it is useful, and it should be on-line by the time you read this:

http://webwatcher.remmie.com

From a functionality perspective, we would like to create more types of agents, and would want to be able to run a permanent version of a user space on a user’s machine, as a stand-alone application. With that in place, it would also make sense to make the agents or the agent spaces a little smarter, so they can run on the user’s machine when he or she is on-line. This could off-load the central server substantially.

Currently the ‘displayDialog’ method is embedded in the main ‘OctarineSpace’ interface. For an architecture perspective, we would want to design a more generic mechanism for agents to query the capabilities of the different agent spaces in the system. An agent could then pick an agent space that can service its needs, and go there. Upon arrival, it should have a simple, yet dynamic mechanism to access the required capability.

CHAPTER 8 Conclusions

We have reached the final stage: the conclusions. We first revisit the goals we set in section 1.3, to see whether we succeed in doing what we set out to do. Then, in a final conclusion, we state the most important results from our discussion. We end with some future directions, showing what we could end up with if we pursue the mobile agents idea even further.

8.1 Goals revisited

In Chapter 1, we established a number of goals, or, more accurately, one goal and a number of sub goals. These were:

> We want to present *mobile agents* as a *distributed application architecture*, as a paradigm for building software systems running on interconnected computers. We want to present *everything agent specific necessary to build such an application*, so we discuss what agents are, what mobile agents are, how a (distributed) software system can be designed and modelled using agents, and how an agent based system can be implemented.

Have we reached these goals? We will look at each piece of these goals in turn:

> We want to present *mobile agents* as a *distributed application architecture*, as a paradigm for building software systems running on interconnected computers.

This goal is addressed in chapter 5, where we show mobile agents as a distributed computing paradigm, and compare it with several well-known distributed computing paradigms. This chapter also lists the most important advantages and disadvantages of mobile agents, and shows several sample architectures for distributed applications based on mobile agents.

> We want to present *everything agent specific necessary to build such an application...*

This is of course a fairly bold statement, and impossible to satisfy if taken to its most general form. However, we worded it like this to emphasise that we did not just want to have a theoretical discussion on agents, but to take it all the way to the implementation level. This is elaborated on in the four sub goals that follow. Indirectly, this goal is satisfied in chapter 7, which shows the construction of an agent-based application from a general idea all the way to the implementation.

> …we discuss what agents are…

This is covered in detail in chapter 2 and chapter 3. Perhaps in the light of the overall goal, a little elaborate. However, there, are two obvious aspects to mobile agents, the mobility, and the ‘agentness’. In chapter 2 and chapter 3, we emphasise the agent aspect, which thus show the possibilities of autonomous agents, and especially the intriguing possibilities of intelligent agents.

> …what mobile agents are…

We introduce mobile agents in chapter 2, and the full implications of the mobility aspect are covered in chapter 5.

> …how a (distributed) software system can be designed and modelled using agents,…

We cover the modelling of agent based systems extensively in chapter 4, where we introduce several enhancements to the UML modelling language to facilitate modelling using agents. The distributed aspects of this, so distributed systems based on agents, and mobile agents, are discussed in chapter 5.

> … and how an agent based system can be implemented.

How agents and mobile agents can be implemented is covered in chapter 6, which also highlights that unless you build everything yourself, it is not yet possible to get create mobile agents in their most general form.

8.2 Final conclusions

We have looked at both agents and mobile agents in detail:

agents
The term ‘agent’ in software has a very broad meaning. We concluded that an agent is an autonomous piece of software that perceives, reasons, and acts. This is fairly abstract concept. To make it more concrete, you can give a certain role to the agent, and that role might vary wildly. ‘Agents’ is more a perspective, a way of looking at and thinking about software a certain way. From a software architecture perspective, an agent is just a computer program. It does become interesting however, when you think of a software system consisting of several agents, and use that concept to design a larger

software system in a certain way. It makes sense to think of a software system in terms of agents if the system needs to behave (partially) autonomous and thus asynchronous with respect to its users.

Because agents are autonomous, they need to make decisions on their own. This makes it very logical to use artificial intelligence techniques for the agent’s reasoning mechanism. We’ve illustrated how the BDI (Beliefs, Desires, and Intentions) formalism can be used to create rational agents that act quite convincing in their environment and towards other agents.

mobile agents
In contrast, mobile agents are almost purely a software architecture construct. The concept consists of two parts: the mobile agents themselves, and mobile agent spaces, which are a hosting environment for the mobile agents. Building a software system using mobile agents substantially changes the way an application behaves on a computer network. Instead of having a fixed, predetermined place where certain code executes, a system build using mobile agents can freely choose (within limits) where certain code is executed.

A simple gain from this is that it is possible to dispatch an autonomous task (the agent) to another computer, and in the mean time the originating client can disconnect and perhaps reconnect somewhere else. Just like a normal agent, the mobile agent by itself decides what to do and now also where to go.

A more advanced gain is that it is possible to optimise execution based on the current network configuration. Agent spaces can be local access points to diverse sources of data, with the agents travelling between those access points to accomplish their task. Those access points can provide very rudimentary access to the data, because the agents can bring their own logic to process the data. It is almost the mirror image of traditional distributed computing. Instead of systems build to lug around large amounts of data for remote processing, mobile agents move small amount of code and data around for local processing.

An additional interesting observation is that where it doesn’t make sense to normal agents for an application with synchronous (user) interaction, it could still make a lot of sense to architect such a system using mobile agents. We could make (part of) the user application a mobile agent, and have it move to the appropriate machines to do its work, and then come back.

8.3 Future directions

We have covered many fairly new topics, most of which are currently actively researched. This gives us plenty of opportunities to suggest future directions. For instance, *emergent behaviour* in adaptive multi-agent systems, where the system as a whole starts to show behaviour that was not ‘put in’ there, is a fascinating topic. We will restrict ourselves to a few future directions that are in a direct line with the main topic: mobile agents as a distributed computing architecture:

8.3.1 Agent programming languages

In chapter 6, we've shown how we can implement agents and mobile agents using existing technology and programming languages. Unless you use one of the (currently) proprietary and non-general agent languages, you need to use some sort of framework. While this works, this is similar to writing an object-oriented program in a procedural language like C. As such, it requires much more discipline and understanding by the programmer. In addition, agents need an asynchronous communication method, which is not really embedded at the language level in current programming languages. In the case of mobile agents, we've seen we cannot move a thread in its current state to another machine. This means we cannot get the complete semantics (seamless migration) of the agent paradigm. This could also be solved by language support for mobile agents.

Some first initiatives have been taken in this step, with the first proprietary (mobile) agent scripting languages and for instance an agent programming language in [47]. We would like to see agent support in (a dialect of) Java or C++.

8.3.2 Asynchronous computer interaction

Most people interact with computer programs in a synchronous way. You issue a command, wait, and view the result. In contrast, humans often interact with other humans in a largely asynchronous fashion. If you ask someone to do something for you, your rarely sit around and wait for this person to complete this task. Instead, you go and do something else yourself, and rely on the person to get back to you with the results. If it takes too long, you would inquire about the task with the other person, who would be able to tell you how far he or she currently is in completing the task.

Agents allow us to have the same interaction model with a computer. However, this might take quite some change on the human side of things. We would have to learn to 'trust' the computer, and we need to find a way to instruct the agent, to give the agent some high-level description of the task you want accomplished. In turn, the agent should be able deduce what you mean, and figure out what assumptions were implicit in your request, and how it should take care of the details of your request. All in all, plenty of room for investigations into agent architectures, human-agent communication, and human-computer interaction.

8.3.3 Improving performance with mobile agents

We've shown in section 5.4.2 how mobile agents can give a performance gain by executing a distributed computation more efficiently. This of course almost screams for a benchmark test, comparing the same application using a 'conventional' architecture to the same application using mobile agents. Assuming the mobile agent version gives better performance, it would also be interesting to compare the mobile agent version to a fully (statically) optimised version of the application, to see how much overhead the mobile agent version brings with it.

Another way to improve performance is using *caching*, keeping a copy of frequently needed data 'near' an application. Whereas normal systems can be are optimised using strategically placed caches in various places in the system, a mobile agents system could (additionally) cache the agent code for optimal execution. Since code almost by definition changes a lot less often than data, and the amount of code generally much less than the amount of data such a cache is much more efficient. If we assume code is cached throughout the system, then, given a suitably long operation time, we can ignore the effort lost for moving around code.

8.3.4 Hiding mobile agents

As introduced, mobile agents are yet another way to deal with the complications of distributed computing. Mobile agents gives a system designer additional flexibility in designing the system, and at the same time offers an additional way to improve performance over other distributed computing architectures. But mobile agents still doesn't solve many issues associated with distributed computing. Without help by the surrounding system, an agent programmer still has to decide when to move the agent. A first step in the right direction could be an *agent application server*, which monitors executing agents and dynamically move them near the data they need.

We think the best thing would be to completely hide all the distributed computing complexity from the programmer, and have the system take care of it. We could build a very clever (Java) virtual machine that makes a set of networked computers look like one single *network virtual machine*. Data is accessible from different physical nodes, but only the network virtual machine has to know where. The threads in such a system are implemented as mobile agents, which means they can take their execution with them and continue elsewhere. Statistical analysis on the executing code could dynamically determine the optimal execution strategy, which of course can change depending on network status. If we take this to the limit, we get a sort of network version of pipelining and branch-prediction as used in modern microprocessors, not to optimise a single processor, but a whole network of them. With this in place, we could truly say: 'The network IS the computer.'[20]

[20] This phrase is used by Sun Microsystems to describe why we should push for open standards, why Java is so great, etc.

References

Note: The titles with a * behind them also appear in [1].

[1] Michael N. Huhns and Munindar P. Singh (editors); *Readings in Agents.* Morgan Kaufman, San Francisco, USA 1998. http://www.mkp.com

[2] Stuart J. Russell and Peter Norvig; *Artificial Intelligence; A modern approach.* Prentice-Hall International, 1995.

[3] Lewton to Ilsa in *Discworld Noir*; GT Interactive, London.1999 http://www.gtgames.com

[4] Klaus Fisher, Jörg P. Müller and Markus Pischel; *A pragmatic BDI Architecture.* Intelligent Agents II: Agent Theories, Architectures and Languages. pages 203-218. Springer-Verlag 1996.*

[5] Anand S. Roa and Michael P. Georgeff; *Modeling rational agents within a BDI architecture.* Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning. pages 473-484. 1991.*

[6] J. Michael Straczynski; *Intersections in Real Time.* Babylon 5 episode 4.18. PTN CONSORTIUM 1997. http://www.babylon5.com

[7] Joseph Bates, A. Bryan Loyall and W. Scott Reilly; *An Architecture for Action, Emotion, and Social Behavior.* Artificial social systems: Fourth European workshop on Modeling Autonomeous Agents in a Multi-Agent World. Pages 55-68. Springer-Verlag 1994.*

[8] Ridley Scott (director) Philip K. Dick(original novel), and Hampton Fancher; *Blade Runner,* Warner bros. 1982. http://us.imdb.com/Title?0083658

[9] Alan Pope; *The CORBA Reference Guide.* Addison Wesley 1997.

[10] Nathaniel S. BorenStein. *Email with a Mind of its Own: The Safe-Tcl Language for Enabled Email.* Proceedings of the IFIP International Working Conference on Upper Layer Protocols and Architectures. (ULPAA). pages 389-402. 1994.*

[11] E. H. Durfee, D. L. Kiskis and W.P. Birningham; *The agent architecture of the University of Michigan Digital Library.* IEE Proceedings-Sofware engineering, Vol 144, No. 1, Feb. 1997, pages 61-71. 1997.*

[12] Justin Cassel, Catherine Pelachaud, Norman Badler, Mark Steedman, Brett Achorn, Tripp Becket, Brett Douville, Scott Prevost and Mathew

Stone; *Animated conversation: Rule-based Generation of facial expressions, gesture and spoken intonation for multiple conversational agents.* Proceedings of the ACM SIGGRAPH Conference. Pages 413-420. Association of Computing Machinery, Inc. 1994.*

[13] Nicolas Negroponte; *Being Digital.* 1995.

[14] Christopher M. Bishop. *Neural Networks for Pattern Recognition.* Oxford University Press, Oxford, New York, USA 1995.

[15] B. Chaib-draa; *Industrial applications of distributed AI.* Communications of the ACM, pages 49-53. Association of Computing Machinery, Inc. 1995.*

[16] Jyi-Shane Liu and Katia P. Sycara. *Multiagent Coordination in Tightly Coupled Task Scheduling.* Proceedings of the First International Conference on Multiagents Systems. Pages 181-188. American Associating for Artificial Intelligence. 1996.*

[17] Henry Kautz, Bart Selman, Michael Coen Steven Ketchpel and Chris Ramming; *An Experiment in the Design of Software Agents.* Proceedings of the National Conference on Artificial Intelligence. pages 438-443. American Association for Artificial Intelligence. 1994.*

[18] Brian A. Stone and James C. Lester; *Dynamically Sequencing an Animated Pedagogical Agent.* Proceedings of the First International Conference on Multiagent Systems. pages 181-188. American Association for Artificial Intelligence. 1996.*

[19] http://www.newhorizons.org/trm_intelligence.html, New Horizons for Learning, Seattle, WA, USA, 1997. http://www.newhorizons.org

[20] Ramesh A. Patil, Richard E. Fikes, Peter F. Patel-Scheider, Don McKay, Tim Finin, Thomas Gruber and Robert Neches; *The DARPA knowledge Sharing Effort: Progress Report.* Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning. pages 77-787. 1992.*

[21] Mark E. Cutkosky, Robert S. Engelmore, Richard E. Fikes, Michael R. Genesereth, Thomas R/ Gruber, William S. Mark, Jay M. Tenenbaum and Jay C. Weber; *PACT: An Experiment in the Integration Concurrent Engineering Systems.* IEEE Computer 26(1), pages 28-38. 1993.*

[22] Ebay Inc. http://www.ebay.com

[23] Terry Pratchett; *The colour of magic.* Colin Smythe Limited, Great Britain 1983.

[24] Brent Sommers; *Agents: not just for Bond anymore.* JavaWorld April 1997. http:://www.javaworld.com/javaworld/jw-04-1997/jw-04-agents.html

[25] Bill Venners; *Solve real problems with aglets, a type of mobile agent.* JavaWorld May 1997. http://www.javaworld.com/javaworld/jw-05-1997/jw-05-hood.html

[26] David Chess, Benjamin Grosof, Colin Harrisson, David Levine, Colin Parris and Gene Tsudik, *Iterant Agents for Mobile Computing.* IEEE Personal Communications 2(5) pages 34-49 1995.*

[27] Dag Johansen, Robbert van Renesse and Fred B. Schneider; *Operating System Support for Mobile Agents.* Proceedings of the 5th IEEE Workshop in Hot Topics in Operating Systems. IEEE Computer Society Press, Washington D.C., USA 1995.*

[28] Daniela Rus, Robert Cray and David Kotz; *Transportable Information Agents.* Proceedings of the international conference on Autonomous Agents, pages 228-236. ACM Press, NewYork, USA 1997.*

[29] David Chess, Colin Harrison, and Aaron Kershenbaum; *Mobile agents: Are they a good idea?* IBM Research Report, RC 19887, 1994.

[30] Amazon.com. http://www.amazon.com

[31] Jeffrey S. Rosenstein, Gillad Zlotkin; *Designing conventions for automated negotiation.* AI Magazine 29-46. 1994.

[32] Mark R. Cukosky, Robert S. EngelMore, Richard E. Fikes, Michael R.Genereseth, Thomas R. Gruber, William S. Mark, Jay M.Tenenbaum and Jay C. Weber; *PACT: An Experiment in Integrating Concurrent Engineering Systems.* IEEE Computer 26(1), pages 28-38 1993 *

[33] Yezdi Lashkari, Max Metral, Pattie Maesl; *Collaborative Interface Agents.* Proceedings of the National Conference on Artificial Intelligence. Pages 444-449. 1994*

[34] James Rumbaugh, Ivar Jacobson, Grady Booch; *The Unified Modeling Language Reference*. Addison-Wesley 1998.

[35] Joseph Schmuller; *Sams Teach Yourself UML in 24 Hours*. Sams Publishing 1999.

[36] Mario Tokoro The society of obects. In Addendum to the Proceedings of the International Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA). Pages. 3-11. 1993.*

[37] General Magic. http://www.genmagic.com

[38] IBM Japan's aglets. http://www.trl.ibm.co.jp/aglets/

[39] The Object Management Group. http://www.omg.org

[40] Randy Fox; *MASIF Revision*. The Object Management Group 1999. http://www.omg.org/cgi-bin/doc?orbos/98-03-09.pdf

[41] ObjectSpace Voyager. http://www.objectspace.com/products/prodVoyager.asp

[42] Graham Glass; *Overview of Voyager: ObjectSpace's Product Family for State-of-the-Art Distributed Computing.* ObjectSpace 1999. http://www.objectspace.com/products/documentation/VoyagerOverview.pdf

[43] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides; *Design Patterns: Elements of Reusable Object Oriented Software.* Addison Wesley Longman 1994.

[44] *Lotus Domino*. Lotus Development Corporation 1999. http://www.lotus.com/home.nsf/welcome/domino

[45] Philip R. Cohen, Adam Cheyer, Michelle Wang, Soon Cheol Baeg; *An open agent architecture*. In Proceedings of the AAAI Spring Symposium on Software Agents. 1994.*

[46] *Enterprise Java Beans*. Sun Microsystems, Inc. 1995-2000. http://www.javasoft.com/products/ejb/

[47] Yoav Shoham; *Agent-oriented Programming*. Artificial Intelligence 60(1) pages 51-92. 1993.*