

A lightweight agent framework for interactive multi-agent applications

Presented at PAAM99, London, April 20-21, 1999

Mikael Kindborg
mikki@ida.liu.se

Johan Åberg
johab@ida.liu.se

Nahid Shahmehri
nahsh@ida.liu.se

Department of Computer and Information Science
Linköping University, S-581 83 Linköping, Sweden
Phone: +46-13-281000
Fax: +46-13-282066
Web: www.ida.liu.se/~mikki/PAAM99

Abstract

Usability issues are traditionally associated with user interfaces rather than with agent frameworks. We argue that the models and metaphors used in a framework will affect the thinking of the developer and influence the application design. Therefore, usability is of central importance for successful software development and for reducing development and maintenance costs. We discuss the design and implementation of a lightweight agent framework for interactive multi-agent applications. A lightweight framework is advantageous for distributed interactive applications, for instance applications running on hand-held devices with limited memory. The design is based on minimalism and simplicity. We present the result from a usability study of the framework, where issues such as learnability and attitude have been evaluated. The study shows that minimalist design principles are useful for achieving understandable and navigable frameworks.

1. INTRODUCTION

We have used a minimalistic approach for the design and implementation of a lightweight Java-based agent framework for multi-agent applications. Our design goal is a small and simple framework that is easy to learn and use, and at the same time generic and extensible. The framework is designed with distributed interactive applications in mind, that is systems where many users interact and communicate with each other. This is an increasingly popular type of software. Internet applications for communication, such as The Palace [3] and ICQ [15], are commonplace, and experimental systems such as Nametags [2] are getting increased attention. Personal communication devices such as mobile phones, personal digital assistants, hand-held games, et cetera, are also developing rapidly. In the near future, jewellery with built in colour displays and sound might be available, as might credit card sized communication devices. For these types of systems, code and data storage must be kept small and efficient. Lightweight operating systems and application frameworks are needed due to the limitations in memory capacity and processing power.

Communication is essential for distributed interactive software. There are many approaches to distributed computing, ranging from low-level protocols and sockets, to higher level solutions like remote method invocation, Linda systems such as JavaSpaces [20], and communicating agents. In the agent model communication is expressed in a clear and explicit manner in the form of message sending and receiving, or even travels for mobile agents. This is a conceptual advantage for systems where many agents, both human and software based, communicate with each other, since the same concepts can be used at all levels of an application (see figure 1). Moreover, the agent model supports the idea of autonomy, which is useful for applications that require processing going on independently of the user's actions. An example is social communication applications where avatars seek contact and interact with each other even when their corresponding user is not logged in.

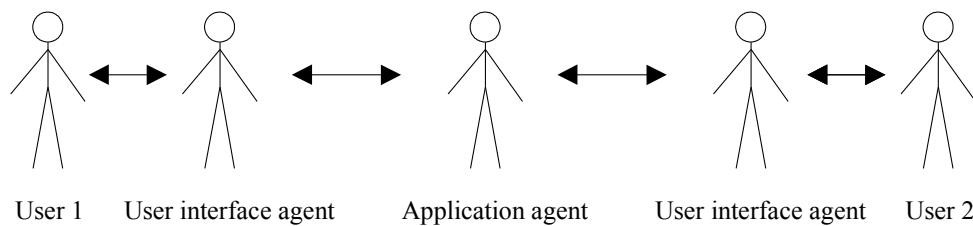


Figure 1: Modelling communication as autonomous agents sending messages to each other can be applied to all levels of an application, both to the user interface level and to the application level. In addition, agents can be used at the data access level (not shown here).

The methodology we have used is based on prototype development and evaluation. First, we designed a generic agent architecture based on the requirements of two different application scenarios. Second, a programming framework based on the architecture was implemented and tested in various applications. Third, a usability evaluation of the framework was done using methods commonly employed for user interface evaluation.

In the remainder of this paper, we discuss design and implementation issues, give examples of applications, and finally present the evaluation results.

2. DESIGN ISSUES

Which issues are important when designing an agent framework? Technical aspects such as performance and compatibility are frequently discussed. However, usability issues such as overview and navigation are not mentioned very often. Usability is traditionally associated with human-computer interaction, but we believe that usability aspects are important also for programmers. We have used the REAL model [14] for usability, which stands for Relevance, Efficiency, Attitude, and Learnability. While all these factors are important, attitude is worth special attention. The fun-factor should not be underestimated. If the developer has a positive attitude towards the tools she is using, increased work satisfaction is more likely to be achieved, and consequently the quality of the job should be positively affected.

In this section, we discuss selected design issues and principles that affect the usability of an agent framework. We will look at simplicity and minimalism, consistency and isomorphism, and finally metaphors and naming. These principles have influenced the development of the agent framework described in this paper, as discussed in section 4. Measuring these aspects in an objective and quantifiable way is difficult. Instead, we have used subjective and qualitative methods. See section 9 for further details.

2.1 Simplicity and minimalism

Good design is generally focused on a specific goal. Compromise often results in poor performance and poor usability. Consider a sports car and a truck as an example. They are both examples of good and useful design because of strong focus on their primary application. It would most likely be less successful to compromise, and try to design a vehicle that both can transport heavy cargo in a reliable way and is fast and fun to drive. Many user interfaces suffer from the designer trying to squeeze in as much functionality as possible. The result is often that the user ends up confused and frustrated.

Simplicity means that concepts should be as simple and as straightforward as possible. Minimalism means that the number of concepts should be as few as possible. In practice, this can be difficult to achieve. Reality is not simple, but diverse and complex. When designing a class library or framework using these principles, one should be careful to not include features that are irrelevant or rarely needed. Every extra function or feature will compete with the relevant functions and can blur the programmer's conceptual model.

Simplicity and minimalism are closely related to consistency. It would for instance be inconsistent and unnecessarily complex to use multiple concepts to represent the same phenomenon. Interestingly, some of the most elegant programming languages, such as Scheme [1] and Smalltalk [8], are based on a consistent and minimalistic design.

2.2 Consistency and isomorphism

Consistency means that concepts and relations should be coherent. Naming, for instance, should be consistent within a framework. Isomorphism means structural similarity. Isomorphism between structures exists when relations and properties of the structures match [13]. An example of isomorphic structures is the reality and the map. Isomorphism can be thought of as consistency between structures. The main advantage with consistent and isomorphic structures is that they are easier to understand and use than disparate structures. In an isomorphic layered model, concepts and patterns are transferable and traceable across different levels of abstraction. In such a model, it will be easier to understand the structures, and to apply knowledge of one structure to the other structures. In the following, we discuss isomorphism across three different dimensions, architectural levels, application layer levels, and design levels.

Architectural levels. The kind of isomorphism which is the focus for this work is architectural isomorphism (see figure 2). The framework architecture can have a strong impact on the application architecture. A mismatch can cause frustration and can result in an awkward application design and implementation.

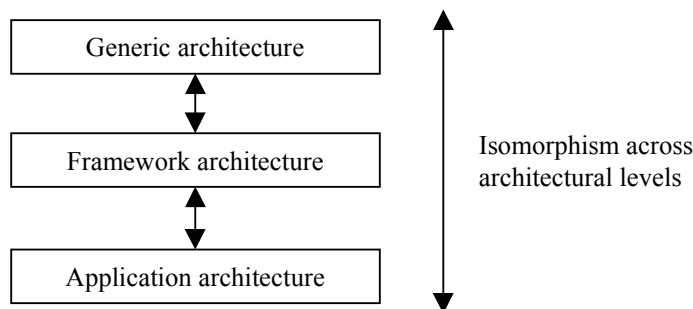


Figure 2: The framework can be viewed as an instantiated generic architecture. Concepts and patterns should be isomorphic across levels, and naming should be consistent. We believe that agents are a useful concept at all architectural levels, just as objects are.

The ideal situation is a match between concepts and relations across all levels. Since each application has its specific demands, this can be difficult to achieve. Designing a framework that attempts to satisfy the need of all potential applications will likely result in a poor compromise. However, within a specific domain, or for a certain application, isomorphism can be achieved. An additional issue is balance between levels. A very simplified generic architecture might not be beneficial, since the application then has to handle all the complex aspects of a problem.

Application layer levels. It is common to structure an application into layers, such as interaction, application logic, and data access and communication. Agents can be used to model concepts in all these layers. There can for instance be interface agents, application logic agents, and data access agents. Users can also be thought of as agents, as shown in figure 1. An extreme degree of isomorphism where for instance the user interface is entirely agent based, should not be a goal in itself, however. An application can benefit from using agents internally and still employ a standard graphical user interface. Similarly, adding agents to one or more of these layers can be used to “agentify” existing applications.

Design levels. In object-oriented system development, objects are used from analysis through design to implementation. The idea is that the same concepts and patterns should be applicable throughout the development process. It should be possible to trace concepts used in the implementation back to the conceptual model. Agents should be used consistently when developing the conceptual model, the design model, and the implementation model. For example, agents identified in the conceptual model should map to agents in the actual implementation. Use of isomorphic design can make it easier for people involved in the development process to recognise and understand concepts and models at various design levels.

2.3 Metaphors and naming

When learning a new programming framework, the developer must build a model in her mind of the concepts and structures that make up the framework. To reduce the mental effort of this process, metaphors and analogies can be used. Metaphors and metaphorical names can help us form a better understanding of the intangible matter software is shaped from.

Metaphors originate from both abstract and concrete domains, and are used frequently at every level of software design. Examples include mathematics (variable, function), biology (neural network), buildings (window, pipe, socket), offices (desktop, wastebasket, file, document, folder, protocol). Concepts such as file, pipe, and stream are nowadays seldom thought of as metaphors, but are used and understood as software concepts. In this sense, they have ceased to be metaphors.

Agents can be thought of as a metaphor for design and programming. Agent architectures and frameworks often use metaphorical names. One example is Odyssey, a mobile agent framework that uses concrete, real world metaphors such as place, ticket, and worker [6]. The agent metaphor combines several useful concepts into one high-level concept. Agents are a higher level concept than objects because they have additional capabilities, such as the ability to communicate, act autonomously, reason and learn. By providing support for agent capabilities a programming framework could reduce application complexity, and thereby simplify development and maintenance.

Naming is an essential aspect of any system. It is well known that it is non-trivial to find good names for classes, methods and variables, when writing a program. The names we choose can

either improve the understanding or distract from the actual meaning. In this respect, programming is a communication act, where a model is communicated to other programmers. It is important that naming is consistent, especially if isomorphism is to be achieved.

Another issue is the level of abstraction. Are concrete metaphors and names better than abstract ones? Metaphors can help our initial understanding, but there is a risk that we make the wrong associations and form an incorrect conceptual model. Semantic mismatch [12] is a problem that can occur when trying to invent a real-world metaphor for a model where some concepts have no or poor correspondence to the original world.

3. ARCHITECTURE

In this section, we give an overview of the framework architecture. To illustrate the isomorphic aspects, we describe the architecture at the generic level, at the framework level, and at the application level. Framework design issues are discussed in section 4. The implementation is discussed in section 5.

3.1 Generic architecture

The framework is based on a generic agent architecture that is intended to be flexible, extensible, simple, and small. The architecture provides concepts and patterns for multi-agent applications. The following are the main components:

- **Agent.** An agent has a unique name. It can send and receive messages and it can act autonomously.
- **Message.** Agents communicate by sending messages to each other.
- **Resource agent.** A resource agent does not perform any actions on behalf of the user; it is a resource that other agents can use. One example is a login agent. Another example is a dictionary agent. A resource agent can also represent a file system or a database. These agents generally do not act autonomously.
- **System agent.** A system agent manages a system of agents. It keeps track of the agents in the system, handles agent communication, and provides support for agent autonomy. Since the system agent is an agent, it can contain other system agents in a recursive tree pattern.

The relationships among these components are shown in figure 5.

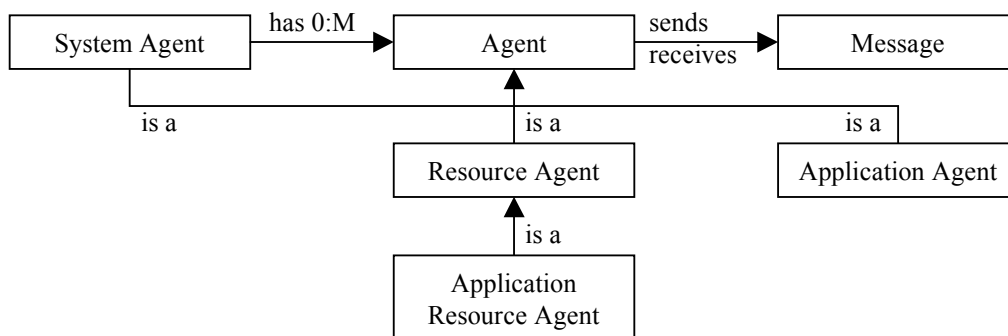


Figure 5: Taxonomy and relationships among concepts.

3.2 Framework architecture

A framework is a supporting structure. A common definition of a framework is “a reusable, ‘semi-complete’ application that can be specialised to produce custom applications” [4]. A framework consists of components and patterns for component usage [11].

The framework we have implemented can be viewed as a specialised or “instantiated” version of the generic architecture. The framework components and patterns closely match those of the generic architecture. Main agent properties supported are agent communication and agent autonomy. There is no support for knowledge representation, reasoning or learning. This can be added by the application, either as new agent classes or as application classes. The framework does not contain any resource agents. The services needed should be added by the application.

The framework is implemented as a set of Java classes. There are in total eleven classes, four public classes that are used by applications, and seven classes that are used internally by the framework. The classes intended for application usage are:

- **Class Agent.** This is the super class of all agents. Class Agent has two methods for sending and receiving messages, `sendMessage()` and `handleMessage()`. Message sending is asynchronous. The format for an agent address is similar to an email-address: `agentID@host:port`
- **Class Message.** A message is a passive container object that holds information about the sender and the receiver of a message, in addition to the message content.
- **Class SystemAgent.** The system agent is an agent that manages other agents and communicates over the network with remote hosts. It has a host address (the IP-address) and a port number. The system agent sends clock-tick messages to the agents in the system at regular intervals which allows agents to act autonomously.
- **Class NotificationAgent.** This class adds highly useful functionality for subscribing to messages. Each subscriber is notified when a message is sent to the notification agent. Notification agents provide a mechanism similar to the Observer (Model-View-Controller) design pattern [7].

The class diagram for the framework is shown in figure 6.

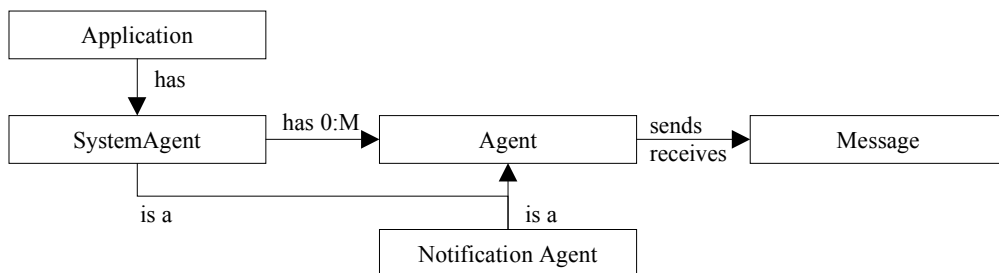


Figure 6: An application should create a system agent, and then create and add additional agents to it.

The notification agent is an addition that is not present in the generic architecture. Notification capabilities are useful for a wide range of applications, which is why this agent is included in the framework. A notification agent informs the subscribing agents when something has happened. This is similar to events in an event driven system, such as Java AWT or Windows. Notifications are extremely useful for updating user interface objects.

3.3 Application architecture

Distributed multi-agent applications involve agents running on several host machines. Each host should have a system agent that handles the communication with that host. In figure 7, an example is given of a generic client-server application architecture based on the framework. In section 6, actual application architectures are described.

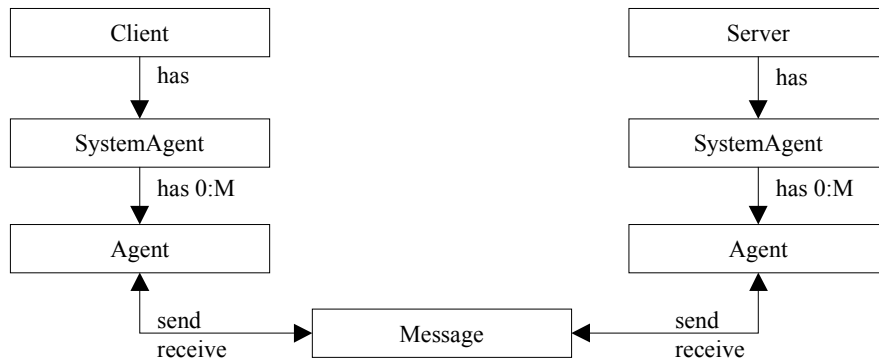


Figure 7: A generic pattern for a client-server application architecture. The system agents handle the actual network communication.

4. FRAMEWORK DESIGN

Here we motivate and discuss the design decisions that have been made. The following is how we have applied the design principles discussed in section 2:

Simplicity and minimalism. The only agent properties supported are communication and autonomy. We believe that this is the basic functionality needed by shared interactive applications. To reduce the number of concepts, every object, except for messages, is an agent. This is why the agent manager is also an agent and is named “SystemAgent”. For simplicity, messages are not agents.

Consistency and isomorphism. We have tried to make the naming consistent for classes, methods, and variables. We have taken great care to maintain a consistent coding style within the framework. The isomorphic properties of the framework are related to the isomorphic qualities of agents as a concept. Agents are generic in the same sense as objects are, and can be used to design isomorphic structures. The framework has been kept general to make it useful for many different applications. It does not include any specific agent types. The price for this is of course less support for the programmer.

Metaphors. We believe that the agent metaphor where autonomous agents communicate by sending messages to each other is easy to understand and to visualise mentally for most people. We have chosen relatively abstract names rather than names based on a real world metaphor. As an example, “SystemAgent” could have been given a more concrete name, such as “Place”, “Town” or “Receptionist”. The reason for choosing neutral and generic names is that this can contribute to isomorphism and a high degree of reusability.

Note that these three aspects discussed above are related to each other. Metaphors, simplicity and consistency, for instance, are closely inter-linked.

Messages and agent mobility are two additional aspects related to framework design:

Message sending. We decided to use asynchronous messages only, since we considered this the simplest approach. Asynchronous messages can be more appropriate than synchronous messages in a distributed environment, due to issues such as network lag and unpredictable response times. There is no wait for a return value, which means that subtasks could execute in parallel, resulting in increased performance. Flexible models for the flow of control are also possible. Multiple agents can, for instance, register as event listeners and act when various events occur. The disadvantage with asynchronicity is that programmers used to the familiar function-call and return-value model have to think differently. The flow of control can also be hard to predict which can make debugging difficult. A general drawback with messages compared to method calls is that there is no compile-time error checking.

Agent mobility. Mobile agents can add increased flexibility and reduce network traffic [24]. A possible extension to the framework is to send any object or agent as a message, using for instance Java serialisation. As with asynchronous messages, clear communication patterns are needed to handle the potentially complex flow of control that mobility can give rise to [22]. A potential problem is the overhead of having an agent bounce back and forth between the local and the remote system. This could result in poor performance if the user wants to control her agent in real time, for example in a game or in a chat world. Since agents have to communicate somehow, messages or method calls are needed also for mobile agents, unless some inventive “mind-meld” could be conceived.

5. FRAMEWORK IMPLEMENTATION

The framework is implemented in Java for portability and popularity reasons, and for the availability of rapid development tools. The first implementation was done over a period of three weeks and took about 50 person hours. However, discussions regarding different design and implementation issues took substantially longer. As with most programming projects, the actual coding is straightforward once the design is evident.

We have tried to make the implementation simple and plain. The number of features has been kept to a minimum. By using basic Java functionality only, like sockets instead of remote method invocation, the required installation effort is minimal and compatibility is achieved with Java versions for hand-held devices, for instance Personal Java.

The size of the framework is shown in table 1. No special efforts have been made to make the code compact. On the contrary, a spacious, easy to read coding style has been used. No optimisations have been done. Error handling is also quite rudimentary.

Number of classes:	11 (4 public, 7 internal)
Compiled code size:	22.7 KB
Source code size:	33.2 KB, 1674 lines (including comments)

Table 1: Size of the framework. Of the seven internal classes, two are optional network classes.

Next, we discuss various implementation issues:

Network communication. Pluggable implementations for TCP/IP (sockets) and UDP (datagrams) have been implemented. Datagrams are suitable for message sending and are

generally faster than sockets. However, the order of delivery is unpredictable and packet loss can occur, especially at high send rates. The network classes are the largest and most complex ones in the framework, largely due to caching of network connections. The socket implementation is 315 lines and the datagram implementation is 382 lines. Many agent platforms provide a name server function that makes the physical location of an agent transparent. We have chosen not to include name server support due to the added complexity involved in set-up and maintenance. There is no arguing that name servers are useful though. It would not be difficult to extend the framework to include this functionality, for instance as a resource agent.

Message handling. A problematic issue with message handling as implemented in the framework, is that an if-statement is needed in the `handleMessage()` method to determine the name of the message and decide the action to take (see section 6). This breaks the object-oriented model. Transparent translation to method calls, for instance by using the Java Reflection API, would be more elegant, but also more complex. Mobile agents could also be used to solve this issue by invoking a polymorph action method on the newly arrived agent.

Use of inheritance versus interfaces. Application agents are created by inheriting from class `Agent`. In this respect, the framework is a white-box framework, rather than a black-box framework [4]. It can be argued that by using an interface for agents instead of a class, any class could add agent capabilities. It would certainly be possible to do this, but it would require a higher implementation effort, since certain basic functionality would have to be implemented by the application agent classes.

Autonomy. Autonomy is implemented using a turn-based (round robin) scheme where agents are sent clock-tick messages at regular intervals. An alternative would have been to run each agent as its own thread. We considered this to be too processing intensive, especially when the number of agents is large. We do not have any empirical data on this however.

Scalability. In the present implementation, a system agent will not scale well. There is a point where the number of agents will be too large, and the system will slow down. However, multiple system agents can be used, on the same host machine or on different hosts.

6. PROGRAMMING EXAMPLES

In this section, we give examples of how to program with the framework.

The message syntax is somewhat similar to KQML [5]. Each message is a string that has a sender, a receiver, a name, and a content. In addition, arbitrary fields are allowed. The name of the message is the “performative” in agent terminology [5], or “selector” in Smalltalk terminology [8]. The format for messages is:

```
(message (to <receiver>) (from <sender>)
         (name <selector>) (content <data>))
```

This message tells Sally that Joe likes her:

```
(message
  (to Sally@maj36.ida.liu.se:4042)
  (from Joe@linal2.ida.liu.se:4042)
  (name tell)
  (content (likes Joe Sally)))
```

The `sendMessage()` method in class `Agent` is used to send messages:

```
agent.sendMessage(message);
```

This method returns immediately since message sending is asynchronous. Because it is kind of cumbersome to create a message object by hand, class `Agent` has an additional `sendMessage()` method with arguments for receiver, message name, and content. This method automatically sets the “from” field to the agent sending the message. In this example the variable “agent” refers to the agent named “Joe” in the above example:

```
agent.sendMessage("Sally@maj36.liu.se:4042",
                  "tell",
                  "(likes Joe Sally)");
```

Incoming messages are received by the `handleMessage()` method in class `Agent`. This is an example of what this method looks like:

```
public void handleMessage(Message message)
{
    if (message.nameIs("tell"))
    {
        // Do something with the content of the message.
    }
    else
    if (message.nameIs("clock-tick"))
    {
        // Do autonomous processing.
    }
}
```

Each agent receives a clock-tick message from the system agent at scheduled intervals. This gives agents a chance to act autonomously.

Applications that are not agent-based, but want to use the communication capabilities of the framework, can use a customised system agent that notifies the application of incoming messages. A subclass of `SystemAgent` can for instance implement the Observer - Observable pattern in the Java class library. Thus, the framework can be used to “agentify” existing non-agent applications by gradually adding agent capabilities to it.

Here follows an example program that illustrates the use of agents and system agents in a typical client-server setting. The server application has an agent that “pings” all messages sent to it. The client application sends a message to the server side ping agent. The port number the system agent will listen to is given in the constructor. Omitting the port number creates a client side only system agent, as shown on the next page in the code for class `PingClient`.


```

/**
 * This agent will exit the application when a message is received.
 */
class PingClientAgent extends Agent
{
    public void handleMessage(Message message)
    {
        System.out.println("Received message:\n",
            message.getSender() + "\n",
            message.getName() + "\n",
            message.getContent() + "\n");

        System.exit();
    }
}

```

7. APPLICATIONS

Here we discuss some of the applications we have developed using the framework. The applications have both shared and unique characteristics, and are intended for different computer platforms (ranging from PC to Java ring). They have varying characteristics regarding complexity, interaction, and communication patterns. The chat requires continuous network connection and users have no persistent representation. Agents in the social world exist on the server even when their respective user is not connected. The emotional sensor might have an irregular network connection depending on the physical location of the user. The chat and the social world have the most intensive interaction, and the sensor has the least frequent interaction.

7.1 A text based chat application

This application is an example of a client-server based chat where people can enter and leave at any time. No login is required for entering the chat. Figure 8 shows the appearance of the user interface. This application is indented for a PC or a PDA.

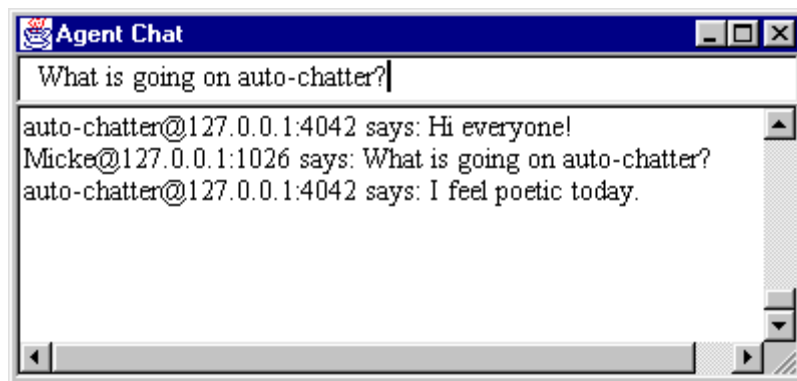


Figure 8: A text based chat for a personal computer. The auto-chatter is an autonomous agent that talks in random sentences.

This application consists of three classes summing up to 208 lines of code (server 32 lines, client 176 lines). Compiled code size is 4.1 KB. The server has a notification agent that forwards messages to all participants in the chat. The server has no additional representation of the users. When adding an autonomous agent to the server we get the structure shown in figure 9. The AutoChatAgent keeps saying random sentences every fifth second using the clock-tick mechanism. The rate of the clock can be adjusted to an interval appropriate for the application. The

clock rate is the same for all agents. If individual rates are needed for each agent, the agents can implement this themselves.

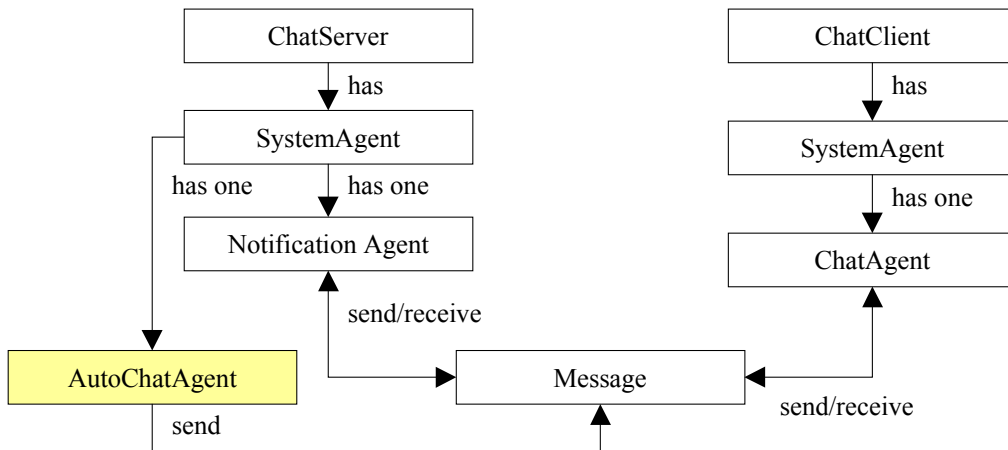


Figure 9: The architecture of the chat application. An autonomous chat robot (AutoChatAgent) is added to the basic architecture as shown in the diagram.

8.2 A social agent world

In this world, a persistent agent, an avatar, represents the user and is active even when the user is not logged in. Avatars seek contact with each other, make acquaintances and gather information for their users. Users can also interact directly with other avatars (see figure 10). This application is indented for a PC or a PDA.

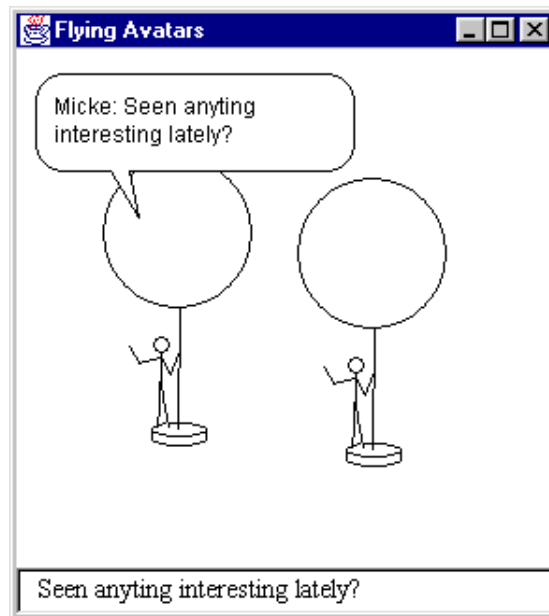


Figure 10: A social agent application designed for a personal digital assistant.

The prototype implementation consists of 1796 lines of code (application 1053 lines, sprite engine 743 lines). Compiled code size is 31 KB. The server has one agent for each user. Every client has a graphical representation of the agents on the server. The architecture is shown in figure 11. In addition to the classes shown, there are two utility classes.

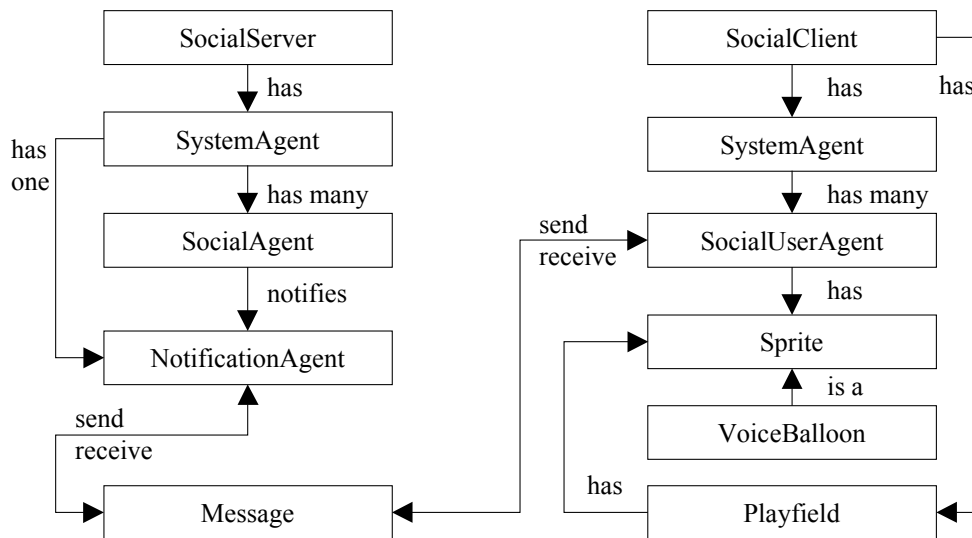


Figure 11: The architecture of the social agent application.

8.3 A remote emotional sensor

The remote sensor functions as an emotional link to someone who is close to us, a relative, a partner or a child. Incorporated into Java based jewellery (a ring, an armband or a necklace) a tiny display shows the emotions of the other person, who is wearing a similar device. See figure 12.

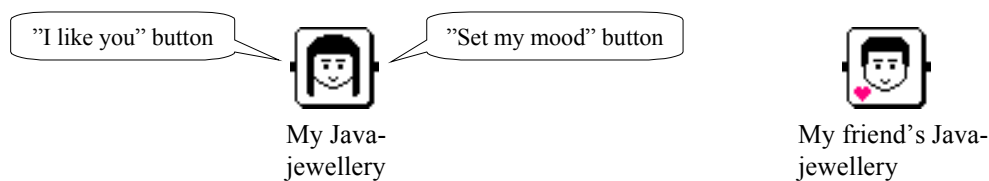


Figure 12: A social application for interactive Java jewellery with a 30-pixel diameter display. By using the buttons on the jewellery the user can change her mood and send a heart to the other person. Inventive input mechanisms could be imagined. Consider for instance using the pearls of a necklace as buttons, or even as a keyboard. A necklace would also make a larger display possible.

This experimental application consists of 547 lines of code. Compiled code size is 9.6 KB. The architecture is based on a client-server model (see figure 13).

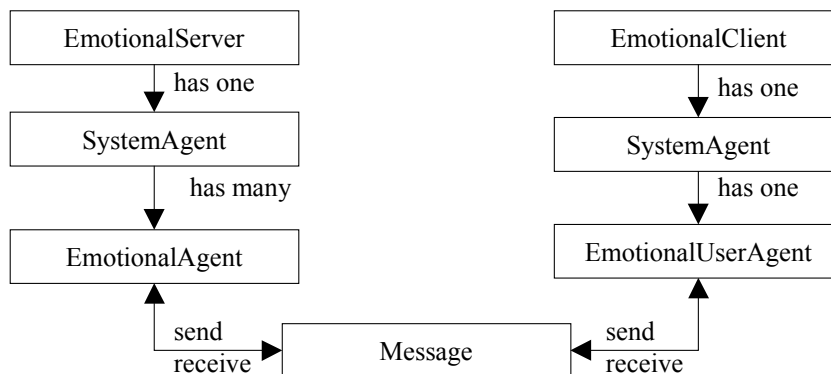


Figure 13: The architecture of the emotional sensor application. The server has one agent for each user.

8. RELATED WORK

Our framework differs from many of the available agent frameworks in its simplicity and small size. The systems we have been inspired by are Aglets [10], Odyssey [6], Concordia [16], and JATLite [19]. All of these are Java-based. In the following tables, a short characteristic of each framework is given:

Our framework	
Agent communication:	Messages (asynchronous)
Metaphor and naming:	Middle concrete/abstract
Autonomy:	Turn-based (round robin)
Number of classes:	11 (11 classes and interfaces, 0 exceptions classes)
Inheritance levels:	2

Aglets	
Agent communication:	Mobility, method calls + messages (synchronous and asynchronous)
Metaphor and naming:	Mix of concrete and abstract
Autonomy:	Threads
Number of classes:	79 (72 classes and interfaces, 7 exceptions classes)
Inheritance levels:	3

Odyssey	
Agent communication:	Mobility, method calls
Metaphor and naming:	Concrete
Autonomy:	Threads
Number of classes:	33 (18 classes and interfaces, 15 exceptions classes)
Inheritance levels:	3

Concordia	
Agent communication:	Mobility, method calls
Metaphor and naming:	Concrete
Autonomy:	Threads
Number of classes:	86 (57 classes and interfaces, 29 exceptions classes)
Inheritance levels:	4

JATLite	
Agent communication:	Messages (synchronous and asynchronous), KQML
Metaphor and naming:	Abstract
Autonomy:	Threads
Number of classes:	54 (52 classes and interfaces, 2 exceptions classes)
Inheritance levels:	6

None of the frameworks provide AI-support such as knowledge representation and reasoning. The metaphor and naming abstraction level is based on our subjective impression. The number of classes and inheritance levels has been counted using the Javadoc class hierarchy (excluding standard Java base classes). It should be strongly emphasised that these numbers are not a good measurement of the usability of the frameworks. They are included for comparative purposes only. Odyssey has been an inspiration in its minimalistic spirit and its pure model. The message handling in our framework was influenced by JATLite and Aglets. We have also been inspired by message-oriented middleware [21]. Three of the above frameworks are based on mobile agents. This is a technical difference, but also an important conceptual issue. Is mobility a conceptual advantage? Or are messages easier to use? A study of the usability aspects regarding this issue would be interesting and useful for further development of agent systems.

9. EVALUATION

This section describes the results from a usability evaluation of the framework. We believe that it is valuable to evaluate agent tools using human-computer interaction methods since programmers are users too.

9.1 Method

We have used a variation of Jacob Nielsen's heuristic evaluation method [18]. Seven experienced programmers conducted the evaluation. Four were from the industry (computer consulting and game/web development) and three were from the academia (computer science). Their previous programming experience varied from 4 to 20 years. Their previous experience with agents varied from no experience to high experience. The average time spent on the evaluation varied between 1 and 3 hours (in one case 15 hours). The evaluators were given questionnaires where they rated usability aspects of the framework. They had access to a web site with an introduction to the framework, javadoc-generated documentation, source code, and programming examples. (This material is available at the web site for this paper.) In the questionnaire, each evaluator ranked different usability issues on a scale from 1 to 10. Each issue also had a field for a comment motivating the choice. The issues concerned the usability aspects: learnability, attitude, efficiency, and relevance. Interviews were also made in some cases, as a complement to the questionnaires.

The advantage with "budget" evaluation methods like this one is that a majority of the central issues can be identified using a small number of expert evaluators. Nielsen mentions five evaluators as an optimal cost-benefit ratio [18]. The selection of questions was made combining Nielsen's usability heuristics [17] and Gustafsson's checklist for user interface evaluation [9].

9.2 Results

The following tables show the result for the usability aspects studied. The questions in the sections of the questionnaire sometimes overlap each other, as do the usability issues themselves, which is why some issues appear in more than one place.

LEARNABILITY

Question	Mean	Min	Max
To learn the framework seems (difficult = 1, easy = 10)	8.9	8	10
To understand the framework concepts feels (difficult = 1, easy = 10)	8.3	6	10
To find classes and methods seems (difficult = 1, easy = 10)	8.7	8	10
To get an overview of the framework structure feels (difficult = 1, easy = 10)	8.0	7	10
The grouping of classes and methods feels (random = 1, coherent = 10)	7.6	6	10
To understand what the framework can be used for feels (unclear = 1, clear = 10)	7.6	5	10
The example programs on the web site were (poor = 1, good = 10)	7.4	5	10
The documentation of the framework was (poor = 1, good = 10)	8.0	6	10

Comments from the evaluators:

- Few classes and concepts make it easy to get going. Naming is clear and natural. Logical naming. Seems to be a clear division of roles between classes. Extremely simple. Not a lot of special functions.
- The class diagram is good for obtaining an overview. Few classes make it easy to get an overview. The underlying agent and communication concepts map directly to the framework.
- Good to start with a really simple example (ping server). Good with small and clear examples. More futuristic, imaginative examples and an AI-example would be nice. Good examples make it easy to understand the framework and what it can be used for. I miss a more problem-oriented example. Screen shots would have been nice, this is also a motivational issue, screen shots stimulates curiosity. Example code is good for learning, but the examples are somewhat uninteresting, since not very many agent aspects are demonstrated.

- It is easy to understand what the framework can be used for, since the services are similar to other network techniques and protocols. There should be more information about what you can do with the framework.
- A bit confusing that SystemAgent and NotificationAgent are agents and not agent managers. Are they agents or agent organisers? Maybe the names are confusing. Why is the system agent an agent? Might have been clearer if the SystemAgent was an independent object (not inheriting from class Agent). Unclear how agents are identified.
- The length of the documentation was right, not too long. A more general introduction to agents was missing. The academic style of the writing feels unfamiliar. Not clear how you do advanced stuff, like intelligent agents floating around in cyberspace. The javadoc-documentation should include description of method parameters. There should be a page that describes the public classes with links to the documentation. Some methods should have been marked as “protected” to avoid unnecessary details. It should be clear which methods are part of the public interface of a class and which are used internally by the framework.

ATTITUDE

Question	Mean	Min	Max
To program using the framework seems (boring = 1, inspiring = 10)	6.7	3	10
The framework structure feels (complex = 1, simple = 10)	7.6	6	9
The number of classes and methods feels (too few = 1, too many = 10, right = 5) <i>Note that this scale is different from the others! Here 5 is optimal.</i>	5.0	3	6
The names of classes and methods feel (poor = 1, good = 10)	8.7	6	10
The naming feels (inconsistent = 1, consistent = 10)	9.0	6	10
The structure feels (inconsistent = 1, consistent = 10)	8.5	8	10
To understand source code of the framework feels (difficult = 1, easy = 10)	8.0	5	10

Comments from the evaluators:

- The framework encapsulates all communication so you can start right away with the fun parts (writing the agents). You only need to know a few methods to get going. Where are the 3D effects :-). A little bit academic, I am more of a cowboy. The framework should contain graphics, for instance a graphical agent base class.
- The framework structure feels natural. The relations feel natural. Feels lightweight. I would have designed it in a similar way. The number of classes is right, too many classes can cause conceptual overload.
- String based messages feels restricting. I would like to be able to send objects as well. It would have been better to assign a SystemAgent to an Agent, because then you would not risk thinking that an agent can exist in multiple system agents.
- Clear naming. Consistent naming. The naming follows Java standards. Very clear source code. Good comments and clear indentation. Some parts of the code are not commented. Instance variables should be marked “private”. Framework should be in a package.

EFFICIENCY

Question	Mean	Min	Max
To develop applications using the framework feels (complex = 1, straightforward = 10)	8.4	6	10
The flexibility of the framework seems (low = 1, high = 10)	7.1	5	10
To extend the framework seems (cumbersome = 1, straightforward = 10)	8.4	6	10
To write programs using the framework seems to be (slow = 1, fast = 10)	8.7	7	10
The reusability of application agents feels (low = 1, high = 10)	7.2	5	10

Comments from the evaluators:

- Good documentation and examples makes it efficient to write programs. Small and lean framework, easy to learn. You seem to get productive quickly. I don't think that reusability is very high for any type of application code, you usually end up rewriting everything in the end.
- Flexible that the intelligence of agents can be added by the programmer. Few required uses of subclassing make it easy to build applications. Message-subscription seems flexible. Good that the source code is available.
- The co-operative multitasking can cause trouble. The handleMessage() method should be marked as “synchronized” to avoid multi-threading problems.
- Error handling and exceptions are missing.

RELEVANCE

Question	Mean	Min	Max
The functionality of the framework feels (low = 1, high = 10)	6.3	3	8
The choice of classes and methods feels (poor = 1, good = 10)	8.4	8	10
The number of application types the framework can be used for seems to be (few = 1, many = 10)	8.0	6	10

Comments from the evaluators:

- Good that network details are abstracted. Remote/local calls should be transparent, however.
- The clock-tick mechanism is restricting. Restricts the potential applications.
- The framework seems useful for applications that need notification messages and broadcasting, for instance monitoring applications. Not good for real-time applications. The framework is useful for all network applications where not optimal performance is required (like action games).
- I don't see that the framework adds very much besides what you can already do in Java. Too few realistic agent examples make it difficult to see what you can do.
- Synchronous calls are needed. Asynchronous messages restrict the usefulness of the framework and the number of application types it can be used for. A standard format for messages and addresses should have been used. XML should have been used for messages. XML would be good for messages since you can handle very complex structures and it is a standard.

9.5 Discussion

The overall structure and design of the framework and the naming got positive ratings. To use the name "agent" for the SystemAgent and the NotificationAgent was noted as a potential source of confusion. Learnability and attitude ratings were high. The number of classes and methods was considered to be right. The major criticism of the framework was related to relevance and efficiency. It is true that our examples do not demonstrate any advanced agent capabilities. There were also comments on the lack of synchronous messages. The clock-tick mechanism was thought of as inadequate in some cases, and potential threading problems were noted. String messages was considered inadequate, and XML was suggested as message format. Graphical capabilities for agents were also asked for, since this usually requires substantial coding efforts.

The budget usability evaluation has been valuable for getting feedback. Using experienced evaluators is an advantage since one can expect qualified comments, but an expert might overlook problems encountered by a novice. A problem with numerical scales is that people interpret them in different ways. A "10" is not the same for all people. We used textual labels on the scales to establish a common frame of reference and to make it clearer what the numbers stand for. The textual comments motivating each rating gave us much more valuable feedback than the numerical ratings. However, the scales provide structure and can serve as a guide for the evaluators, helping them to focus on the central issues.

9.6 Design and evaluation principles

The questions asked in the evaluation are related to the following principles, which are adopted from Nielsen's usability heuristics [17] and Gustafsson's checklist for user interface evaluation [9]. This list is our suggestion for heuristics that can be used for design and evaluation of agent frameworks and similar systems:

- **Overview and navigation.** Is it easy to get an overall view of the framework? Is it clear which functions are available? Is it easy to navigate the framework and find relevant classes and methods?

- **Structure and grouping.** Is it clear how classes and concepts are related to each other? Are classes and methods grouped in an intuitive way?
- **Help and documentation.** Is the documentation useful? Can it be used both for learning and as a quick reference? Are there any examples and example programs that help to quickly learn the framework?
- **Relevance.** Is the functionality provided relevant for the problem at hand?
- **Flexibility and efficiency of use.** Is it cumbersome or straightforward to use the framework in an application? Can the framework be used in different ways depending on the programming task?
- **Extensibility, programmer control and freedom.** Can the programmer extend and tailor the framework? Does the framework place restrictions on the programmer that feels counter-productive?
- **Simplicity and minimalist design.** Are the concepts as simple as possible? Is the number of concepts small? Are there any redundant or less central concepts that should be removed?
- **Consistency.** Are concepts used consistently? Is the naming consistent? Are the same principles used throughout the framework?
- **Attitude and fun-factor.** Is the framework fun to use? Does the design feel “right”? What is the first impression?

10. DISCUSSION AND FUTURE WORK

Are small and simple frameworks better than complex and feature-rich ones? There is probably no single answer to that question. Which approach is preferred is likely to depend on the application requirements. The evaluation shows that simplicity and minimalism are useful design principles for achieving understandable and navigable frameworks. The number of concepts was considered close to optimal by the evaluators. A small number of consistent concepts are clearly a conceptual advantage. The documentation, and in particular example programs, was noted as important in the evaluation. A description of concepts is not sufficient for learning a framework. The actual patterns and idioms are much more central for understanding typical uses.

We intend to continue to work with applications for user to user communication. Hand-held devices running Java are an exciting development. We plan to develop a more advanced version of the social agent world in order to experiment with emotional interaction and user programming of agents. We are also doing work in web based recommendation systems. A system for web shop assistants has been successfully tested on a video shopping site. An interesting area is visualisations of agents, which was suggested by one evaluator. Innovative graphical appearances for individual agents and visualisations of agent structures can be imagined. The latter could be useful for understanding the dynamics and behaviour of an agent system.

Messages versus mobile agents are an interesting area. The message model of our framework was criticised in the evaluation, and we plan to experiment with sending any type of object or agent as a message. An interesting alternative to Java and similar languages are languages with built in mechanisms for communication and mobility support, such as Distributed Oz [23].

We strongly feel that human-computer interaction methods and principles are valuable for developing agent tools and frameworks. Many of the comments we got in the evaluation are not specific to agent systems. As is the case with many new and popular fields, agents can benefit from general principles and methods of software design.

Additional information, example programs, source code, and the material used in the evaluation, are available on the web at: www.ida.liu.se/~mikki/PAAM99.

11. ACKNOWLEDGEMENTS

We would like to thank all persons participating in the evaluation of the framework. This work has been funded in part by the Graduate School for Human-Machine Interaction at Linköping University and in part by Enersearch AB.

12. REFERENCES

- [1] Abelson, Harold and Sussman, Gerald Jay. Structure and Interpretation of Computer Programs. MIT Press. 1985.
- [2] Borovoy, R., Martin, F., Resnick, M., Silverman, B. GroupWear: Nametags that Tell about Relationships. *Proceedings from CHI 98*. ACM Press. 1998.
- [3] Electric Communities. The Palace Web Site. 1998. Available at: www.thepalace.com
- [4] Fayad, Mohamed E. and Schmidt, Douglas C. Object-Oriented Application Frameworks. *Communications of the ACM*. Vol. 40, No. 10, October 1997.
- [5] Finin, Tim, Labrou, Yannis, and Mayfield, James. KQML as an Agent Communication Language. In *Software Agents*. Ed. Jeffrey, M. Bradshaw. AAAI Press. 1997.
- [6] General Magic. Odyssey White Paper. 1998. Available at: www.genmagic.com
- [7] Gamma E., Helm, R., Johnson, R., Vlissides, J. Design Patterns. Addison-Wesley. 1995.
- [8] Goldberg, Adele and Robson, David. Smalltalk-80, The Language and it's Implementation. Addison-Wesley. 1983.
- [9] Gustafsson, Nils-Erik. Checklist for user interface evaluation (In Swedish). Ericsson Utvecklings AB. 1996.
- [10] IBM. Aglets. 1998. Available at: www.trl.ibm.co.jp/aglets
- [11] Johnson, Ralph E. Frameworks = Components + Patterns. *Communications of the ACM*. Vol. 40, No. 10, October 1997.
- [12] Kahn, Ken. Metaphor Design. *Proceedings of the Game Developers Conference*. April 1995. Available at: www.toontalk.com
- [13] Lundequist, Jerker. Design and product development (In Swedish). Studentlitteratur. 1995.
- [14] Löwgren, Jonas. Human-computer interaction. Studentlitteratur. 1993.
- [15] Mirabilis. The ICQ Web Site. 1998. Available at: www.mirabilis.com
- [16] Mitsubishi. Concordia White Paper. 1998. Available at: www.meitca.com/HSL/Projects/Concordia
- [17] Nielsen, Jacob. Ten Usability Heuristics. 1998. Available at: www.useit.com
- [18] Nielsen, Jacob. How to Conduct a Heuristic Evaluation. 1998. Available at: www.useit.com
- [19] Stanford. JATLite Web Site. 1998. Available at: java.stanford.edu
- [20] Sun. JavaSpaces White Paper. 1998. Available at: java.sun.com/products/javaspaces
- [21] Shoffner, Michael. Write your own MOM! Write your own general-purpose, message-oriented middleware. *JavaWorld On-line Edition*. May 1998. Available at: www.javaworld.com
- [22] Tolksdorf, Robert. Coordination Patterns of Mobile Information Agents. *Proceedings from Cooperative Information Agents II*. Paris, France, July 1998.
- [23] Van Roy, P., Haridi, S., Brand, P., Smolka, G., Mehl, M., and Scheidhauer, R. Mobile Objects in Distributed Oz. *ACM Transactions on Programming Languages and Systems*. Vol. 19, No. 5, September 1997.
- [24] White, James, E. Mobile Agents. In *Software Agents*. Ed. Jeffrey, M. Bradshaw. AAAI Press. 1997.