

# **AMETAS**

*Good Migrations.*

## ***AMETAS White Paper Series***

by Klaus Herrmann  
and Michael Zapf

# **On Multi-Agent Applications**

July 2000

Johann Wolfgang Goethe-Universität Frankfurt/Main, Germany  
Department of Computer Science

[www.ametas.de](http://www.ametas.de)

## Multi-Agent Applications

Several White Papers in the *AMETAS White Paper Series* focus on technical details of AMETAS and of writing agents, services or user adapters. When you read those papers you might ask: *But how do I put it all together and form a complex multi-agent application?* This White Paper tries to give you an insight on some principles involved with structuring such an AMETAS application. We do not concentrate on a concrete implementation but rather give a general view on this question which should help the reader to understand the problems of agent application design.

## Identifying Place Users

The first thing you will do when analyzing a task is to identify *real-world* objects which can be modeled as software objects. In AMETAS the question is: *Which kind of Place Users (software objects) do you use to model a task?* The first rule is:

*Not everything is an agent!*

The most common mistake made by programmers that are new to agent-oriented programming paradigm is that they try to model everything as an agent. This results in restricting some components that might need some more power and don't need mobility. Normally, multi-agent applications do not consist of numerous different agent types but rather of one or two different agent classes that are sometimes used to instantiate numerous agent instances.

Entities that are services by nature should be designed as AMETAS services. Components that belong to the user interface should be realized within the user adapter of the application. Especially the last point sometimes poses some conceptual difficulties for programmers. Take for example an application which may be extended by adding new agent classes. Each of these agent classes needs to be configured in its own very specific way. Therefore it provides a custom configuration dialog. Where does this dialog belong? To the agent, you might say, since it is used to configure it. Well, viewed from a very high level this might be true, but what about the agent's inherent restriction to not display any GUI? The configuration dialog is obviously a part of the user adapter's interfaces used for starting agents. Therefore, at a lower level the dialog must belong to the user adapter, not the agent. We will revisit this example later on in this White Paper.

Usually, a multi-agent application consists of a few agent classes, a few services installed on every place and one user adapter installed on the user's place.

## The User Interface

The design of a user interfaces follows the rules known from normal applications and will be determined by customer wishes and designer preferences. However, there are a few things that should be considered when designing a user Interface of a multi-agent application.

**Find a Balance between Agent Awareness and User Annoyance** As we stated in other White Papers, awareness is needed much more than transparency in mobile agent software. However, when it comes to user interfaces, the end user usually does not want

to be overwhelmed by details about agents. Most users prefer to use such an application in the same way as a *normal* one. The designer has to find the right balance in what information about the internals are useful for the user and which details just annoy him. At this level, transparency is good to a certain extent while at the programmer's level awareness is needed.

**Try to Employ Well-Known and Easy-To-Use User Interface Techniques** A similar tip is not to try and use absolutely new and *freaked-out* interfaces which seemingly fit the new paradigm. Users will be confused and this is not what we want.

**Dealing with Errors in Distantly Executing Agents** What happens if one of your application's agents ceases to function on a distant place due to some error? First of all, as we already stated in the White Paper about designing agents, make your agents bullet-proof, i.e. catch and handle every possible exception and implement the `recoverError` method. This way you always have the chance to send back an error notification to the user adapter which can display the error to the user. Nothing is worse than an agent getting lost without the user being notified. If your application is designed for disconnected operation, you might choose to send a messenger agent (see the White Paper on communication) which will wait on the relay place until the user is online again.

**Starting Agents** Agents should be started from within the user adapter only. Do not confuse the user by telling him to use AMAI or AMSI<sup>1</sup> and start agents manually. Starting agents this way will not be very helpful in trying to integrate the user into the application. Use the AMETAS API to start agents from the user adapter's own SPU container or from the agent's SPU container. The difference between these two techniques is that in the first case, the agents have to be packed into the user adapter's SPU file while in the second case they are separated from the user adapter and come in their own SPU file. If you want to build an application which should be easily extensible by adding new agents, you should use separate SPUs for agents. If the agents are fixed and there is no need to add new ones, you might choose to put them into the user adapter SPU file since such an application will consist of one single SPU file and is easier to distribute and install.

**Dynamic Loading of Agent-Specific Classes** We already talked about extending an application with new agent classes. If this is desirable, we often need a way to configure agents that is specifically tailored to their tasks. If we do not have a common configuration interface for all of them, we need to dynamically load the interface for each agent. AMETAS introduces *Signed Class Containers* (SCCs) that can contain any Java classes. A SCC can be imported into a user adapter by calling the driver's `importClasses` method. In this scenario, every agent (or rather their programmer) provides an SCC which contains the classes necessary for configuring it along with its own SPU file. When the agent is to be configured before its start, the SCC is imported into the user adapter and its classes can be instantiated by the user adapter, for example, to display a configuration dialog.

---

<sup>1</sup>Read the White Paper on interfaces for more information about AMAI and AMSI.

## Accessing System Resources

System resources are accessible for agents by using services as stated in our White Paper on service development. Some simple rules for service programming are presented here.

**Don not Implicitly Give Agents Unlimited Access to the System** You might get into trouble doing this. Always remember that every mobile agent is a potential attacker. So limiting system access to a minimum is advisable. Giving agents unlimited freedom by designing a general access service might be convenient but its very dangerous not only for the system but also for you. After all, it was *you* who designed the services and created the security hole. Set proper *Required Privileges* to keep unprivileged agents from using your services. By explicitly requiring any client to have some specific privilege to be able to use your service, you can prevent foreign clients to attack the system by using your service.

**Efficiency Through Multi-Threaded Services** Consider designing your services to be multi-threaded. This might reduce response times and vastly increase performance. The current AMETAS distribution allows only one client request at a time since the event mechanism used to receive requests works this way. If you assign a thread to a client request and immediately return to let the next request get through, you will resolve this implicit serialization of requests. Use a fixed set of worker threads and assign free ones to incoming requests. If, however, your service is expected to receive only few requests in irregular intervals, you might be better off without multi-threading.

**Make Use of Service Parameters for Service Configuration** Services can be configured with service parameters which are given to each service object on initialization. Make use of this possibility to enable administrators to configure your service.

**Avoid Platform-Dependent Services** One of the biggest mistakes committed by most Java programmers is to introduce explicit platform dependencies in their classes. For example using “/” as path separator in file names instead of using `File.separator` will render your service useless on Windows machines. It is amazing how often this mistake can be found in Java software (including my own programs ;-)).

**Find a Balance between Big and Powerful Services and Small but Flexible Ones** Programmers tend to implement one monstrous service that can do anything for the agents of an application. This might be the right way if an application is small, but packing all the functionality needed by a large scale application into one service will result in inflexible behaviour. When you split up such an *almighty* service into several smaller ones according to the service’s different tasks, you have the possibility to replace one of them without touching the others. It also enables you to selectively install only a subset of the application services on a place. However, you pay for this flexibility with increased inefficiency since every Place User consumes system resources just by existing. Again, balance is the magic word.

## Designing Mobile Agents

Especially when designing your first agent application you will be the victim of some general misconceptions about agent software. Efficiency and clever computational schemes are much much more important than in the design of *traditional* software. The reason is that, for example, bandwidth usage and security become central factors where as normal software development has different focuses.

**Don't Overload Agents** Sometimes the programmer has the choice to put certain functions into the agents or to position them in services for more efficient migration. Think carefully whether all the functionality you put into your agents is really needed. Especially when agents tend to migrate rather often this choice could be vital.

**Get Used to the `transient` Statement** Don't hesitate to use the `transient` modifier on the agent's instance members. It might save considerable amounts of bandwidth.

**Consider Data Compression** If your agents collect some kind of information while traveling the net, think of clever mechanisms for compressing this information. These mechanisms can be things like zipping the information or they may involve semantic compression. Remember that semantic compression is one of the strengths of mobile agents. They process data locally extracting the important information and only transport this extracted data. Unreflectedly transporting uncompressed data may cost you dearly.

**Think Twice about the Agent's Migration Policy** Think about your migration policy carefully. A second thought while designing an agent might be worth several useless migrations that will waste bandwidth and eliminate a big advantage of mobile agents. Remember, in many scenarios it is actually a good idea to migrate agents to the data but when we start migrating for every single byte we will end up in *IT-hell*<sup>2</sup>. Experience shows that most agents can do their job by migrating only once or twice. Extended trips through the Internet might be the first thing that comes to your mind when you think of mobile agents, but most cleverly designed agent applications do not need such trips. The following drawbacks result from frequently migrating:

- Bandwidth is wasted and the network easily gets overloaded.
- Hosts are busy sending code around instead of executing it to complete the task.
- Agents become susceptible to getting lost due to transmission errors or host crashes.
- Agents are exposed to attacks more often since their code and state is transmitted over publicly used connections.
- It gets very hard to judge the behaviour of the application.

However, some application might require frequent migrations. Therefore, the needs of the application have to be analyzed very carefully before migration schemes are fixed.

---

<sup>2</sup>I have been there. Believe me, it can be a rather unpleasant place.

## Communication Aspects

As can be read in the respective White Paper, communication among AMETAS Place Users might be a topic many programmers first have to get used to. Sending and retrieving messages is quite different to calling methods.

**Use Names and Groups in Place User IDs** The AMETAS messaging system allows for a simple form of multicast and broadcast (see the White Paper on communication) which is based on Place User IDs. The *name* and *group* fields are of special interest in this respect. Define useful names and groups for your Place Users to facilitate communication. For example, if you feel the need to contact all present agents of a certain group of agents, then you might consider giving a common group identifier to them. When you send a message destined for this group, set all other fields in the Receiver Mask of the message to `null` (wildcard) to multicast the message to all agents of that group. Remember that only those agents of that group will get your message which are present at the local place now or at a later point in time. The name field can be used in a similar manner. It has no deeper meaning and may be chosen freely to serve your purposes.

**Send the *right* Classes in Messages** Remember that objects of classes that are not present in the place's CLASSPATH are not eligible to be send within a message. The reason for this is that AMETAS' security system fundamentally relies on Place Users being loaded via individual class loaders. Moving non-system classes from one class loader to another inside a message will result in an `IllegalAccessException` since the receiver's class loader cannot access the object class' original class loader. If you represent data by using classes only defined in the agent, you will have to convert them, for example, to standard Java classes and do the reverse in the receiver's code.

**Use Events when Communication Starts Getting Complex** Place Users that engage in long lasting, semantically rich conversations with numerous other Place Users should use the *extended event mechanism* introduced by the means of Event Handlers (see the White Paper on events). This mechanism allows communication partners to receive messages in a timely fashion and to conveniently process them in multiple separate Event Handlers if necessary. This may help structuring your Place Users in a way that makes them easier to maintain. Again, it depends on the complexity of communication whether Event Handlers are an unnecessary overhead or a vital means for structuring.

## A General Word about Agent Design Paradigms

The OO world has seen some design methods for OO software come and go. Some proved valuable, others impractical. In the agent community a general recipe for designing applications is still missing. One reason might be the general lack of consensus on the question what an agent is. The diversity of existing agent systems might play a role too. However, science is far from giving a step-by-step plan on how to develop a good multi-mobile-agent application. Experience will show if something like this is achievable. Chances are that people might have to carry on developing their own application-dependent ideas in this area.