ESPRIT Project No. 25 338

Work package D

Autonomous Agents

DD3: Design

ID:	DD3.2	Date:	05/05/98
Author(s):	Nick Taylor, Steve Battle	Status:	
Reviewer(s):		Distribution:	

Change History

Document Code	Change Description	Author	Date
DD3	Baseline version	NT/SB	09/04/98
DD3.1	Additional material added	NT	17/04/98
DD3.2	Changes arising from internal review	NT/SB	05/05/98

1 INTRODUCTION 5

2 BACKGROUND 6

2.1 Software Agents 6

2.2 Definition of an Agent 6

2.3 Agents and Mobility 7

2.4 Mobile Agent Systems 7

2.5 Applications of Mobile Agents 7
2.5.1 Information Retrieval 7
2.5.2 Monitoring Remote Resources 8
2.5.3 Market places and Auctions 8

2.6 The "Personal Assistant" Metaphor 8

2.7 Agent Languages 9
2.7.1 Scripting Languages 9
2.7.2 Scripting Architectures 10
2.7.3 Scripting the Desktop 10
2.7.4 Scripting the Web 10
2.7.5 Scripting for Agents 10

3 REQUIREMENTS AND ANALYSIS 12

3.1 Domain Object Model 12

3.2 Roles of the Domain Objects 13
3.2.1 Administrator 13
3.2.2 Agent Place 13
3.2.3 Personal Assistant 13
3.2.4 Personal Assistant Factory 13
3.2.5 Personal Assistant Directory 13
3.2.6 Task Agent 13
3.2.7 Task Agent Factory 14
3.2.8 Task Agent Directory 14

3.3 Analysis of Use-Cases 14
3.3.1 Create a PA 14
3.3.2 Connect to PA 14
3.3.3 Disconnect from PA 15
3.3.4 List Services 15
3.3.5 Select a Task 16
3.3.6 Initiate a Task 16
3.3.7 Signal to Agent 17
3.3.8 Communicate with Agent 17
3.3.9 Report to User 17
3.3.10 Communicate with User 18
3.3.11 Request Move to Remote AgentPlace 19
3.3.12 Request Move to Local AgentPlace 19
3.3.13 Destroy a PA 20

4 DESIGN 21

4.1 Factories 21

4.2 Traders and Directory Services 21

4.3 Agents 22

4.4 Task Agents 22
4.4.1 Creating and Using Agents 23
4.4.2 Scripting Basics 23
4.4.3 Object Hierarchy 24
4.4.3.1 Missions 24
4.4.3.2 Parameters 25
4.4.3.3 Dynamic properties 26
4.4.4 User Interaction 27
4.4.5 Script-directed Mobility 27
4.4.6 Threading 28
4.4.7 The Java connection 28
4.4.8 The Pizza Factory 29
4.4.9 Migration Paths 30
4.4.10 Goal-oriented Description 30

4.5 Personal Assistants 30
4.5.1 Movement of PAs 31
4.5.2 Uses of the PA 31

4.6 AgentPlaces 32
4.6.1 Auditing Features 32
4.6.2 Starting an AgentPlace 33
4.6.3 Implementing an AgentPlace 33

4.7 High-level Class Diagram 34

4.8 Summary 34

5 REFERENCES 35

APPENDIX A - BNF FOR ECMASCRIPT 37

APPENDIX B - MISSION AND SCRIPTING TAGS 40

APPENDIX C - INTERFACE SPECIFICATIONS 43

1 Introduction

The advent of affordable and lightweight computers, such as laptops and Personal Digital Assistants (PDAs), has allowed people from all walks of life to carry a digital desktop with them when travelling. These devices provide a familiar interface which is probably present on the user's main home or office computer so they feel comfortable and have access to all the tools they are familiar with. The problem some users face however, is that data they may want, be it leisure oriented or corporate, is only available via a network such as the Internet, and accessed through direct manipulation of a forms based interface. This means the user must be connected to the network providing the service in order to use it. Technology has now advanced to the point where mobile communication devices, such as mobile phones, are cheap and readily available. The combination of laptops and mobile phones give the user the ability to connect to a network from almost anywhere but this could prove to be costly for long connection times. Currently, most services on networks like the Internet are provided through forms however, if the network services were to present themselves programmatically then we have the potential to interact with them indirectly through software which does not necessarily require direct manipulation from the user.

Collectively, the Autonomous Agents, User Access, Service Interaction and the Personal Profiles Work-Packages (WPs) will seek to enable, through the use of many emerging and different technologies, applications requiring the behaviour outlined above to be realised. The Autonomous Agents WP in particular will use mobile code technology and a software abstraction known as *agents*. Agents which carry their own code with them are known as *mobile agents*. If these mobile agents are used in a rich framework of interacting components, we can envisage applications that *follow* the user, *gather* information on behalf of the user and *deliver* information to the users digital desktop. This WP, together with the Service Interaction WP and the Personal Profiles WP will provide the framework required to enable these types of activities and implement it in a component form. The components will comprise an environment for hosting agents, tools for building mobile agents and through the Personal Profiles WP, allow the agent access to information belonging to its owner. The Service Interaction WP will provide the facilities to allow service based applications to built and the framework which will allow the agents to interact with them. Meta-descriptions of the services will be investigated as a means to describe the behaviour of a service and how it may be used. A scripting facility will be provided by the Autonomous Agents WP which will allow autonomous, goal oriented software agents to be built. A special agent known as the Personal Assistant will take care of the users information and agents while the user is not connected to the network. Through its interaction with the User Access WP, the PA will present the user's information via a number of different devices.

These components will form the agent framework upon which the pilot applications will build solutions in response to very different requirements; a regional event notification system, a stock portfolio management system and an electronic newspaper. Indeed, the requirements of these pilots influenced considerably the functionality of the agent framework. Through the use of Java as the implementation language, these applications will be able to operate on heterogeneous platforms and so make many portable and non-portable devices capable of providing a communication end-point for the application.

This document will introduce software agents and scripting, then go on to describe the functionality to be provided by the Autonomous Agents WP through the requirements elicited from the pilot applications. A design will be presented for a component based solution which will provide the required functionality together. Interfaces to the various components are stated in Appendix C. The diagramming tools and notations used conform to the Unified Modelling Language specification.

2 Background

In recent years, the term "agent" has been used more and more to describe a multitude of computing related activities up to the point now where it has become a rather loose and vague label. Researchers, commentators and vendors are, to some extent, all to blame for the current confusion surrounding the role of agents in emerging architectures and applications.

It should be noted that this section will focus on the area of mobile agents and not intelligent agents. FollowMe is concerned with issues of mobility and therefore a survey of intelligent agents is beyond the scope of this document

2.1 Software Agents

We are all familiar with the human notion of an agent, some of us come into contact with them every day. Consider an estate agent; they act on behalf of other parties in order to facilitate the buying or selling of a house. The key here is "act on behalf" which implies an act of delegation from the client to the agent. The purchase client delegates the activity of finding suitable properties to the agent who then selects suitable candidates from an available pool, according to the criteria specified by the buyer, and then presents these to the client. In the case of a selling client, the agent takes the details of the property and makes them available to purchasing clients. Often the agent will charge a fee for doing something on behalf of a client; in the estate agent case a selling client is usually charged by the estate agent in the form of a percentage of the selling price, however, a buying client is not normally charged.

The computational agent is very similar to the human notion of an agent in the sense that an entity delegates an activity to the agent which is then expected to carry out that activity. It may also be the case that the agent charges a fee of some description but this is not always the case. A software agent, much like its human equivalent, has to prove its competence in being able to represent a client's interests and thereby gain the user's trust; two issues discussed by Maes [13].

2.2 Definition of an Agent

For the purpose of this document, we will adopt a "weak" or un-contentious definition of a computational or software agent as described in [24] and suggests that an agent is:

· ***autonomous***: an agent has its own goals and mechanisms to achieve those goals.

· ***social***: an agent can interact with other agents and humans through well defined interfaces.

· ***reactive***: an agent is able to perceive its environment and respond to changes in it.

· ***proactive***: an agent is able to not only react to its environment but also to demonstrate goal directed behaviour and can take the initiative where possible to achieve its goals.

As was mentioned earlier, these four abilities define a weak notion of an agent or the basic features most definitions include. There are other ideas an agent can embody that are more contentious such as veracity (the assumption that an agent will not knowingly give false information), benevolence (the assumption that agents do not have conflicting goals and will always try to do what is asked of it) and rationality (the assumption that an agent will act in order to achieve its goals). For a more detailed discussion on the "strong" definitions of agents, see [6].

2.3 Agents and Mobility

Agents which are capable of travelling from machine to machine via networks are referred to as Mobile Agents. Mobile agents embody the four ideas mentioned earlier and add the ability to carry their state and code with them when they move. This leads to a new model for distributed computing where the clients, which are typically static in peer-to-peer and client-server models, have the ability to move between “places” which are abstractions of the physical machines in the network. This allows an agent to migrate to the service (or agent) they are interested in interacting with, and so reduce network traffic during the interaction with the agent needing only to report salient events (if indeed there are any) back over the network to an interested party.

2.4 Mobile Agent Systems

Mobile Agent Systems provide a development environment which allows the construction of applications based upon agents. The first commercially available agent system was Telescript from General Magic [7]. Telescript allowed mobile agents to be created and deployed in abstractions of the physical host known as “places”. The agents could interact with each other and utilise the built in security features for authentication. Although systems using Telescript were deployed, the system never really caught on mainly due to the proprietary language agents were written in and the cost. The emergence of Java which, due to the combination of an easy to use language, virtual machine and byte code definition, fuelled a number of agent system development initiatives and added great value through the freely available Java Development Kit (JDK) and the numerous platforms supported. One of the first Java-based agent systems to emerge was MOLE [16] from the University of Stuttgart; it provides basic facilities such as the concepts of locations, agents, migration of agents between locations, remote execution of methods and messaging.

Many other agent systems are available presently, and more seem to appearing on an almost daily basis. Some of the more well known systems are Aglets from IBM [9], Voyager from ObjectSpace [17], Concordia from Mitsubishi [14] and Odyssey from General Magic [7]. These systems all support the weak notion of agents to some degree and some systems provide services such as security and persistence which agent based applications may use.

Efforts are under way to standardise the interfaces and components of agent systems with the OMG issuing request for proposals for a Mobile Agent Facility (MAF) [18]. Crystaliz, General Magic, GMD FOKUS and IBM have made a joint submission which under consideration. However, due to evolving nature it is not suitable as target specification in FollowMe.

2.5 Applications of Mobile Agents

There is a great deal of research being conducted into the applications of mobile agents (see [24] for more detailed information). The following sections outline some of the application areas of interest.

2.5.1 Information Retrieval

According to Maes [13], the "information overload" situation is with us; there is just so much unstructured information available that we are in danger of being swamped. Maes has suggested that agents could be one way one

way of avoiding this crisis. Off-line retrieval is a promising application area that allows a user to delegate the search for information to a software agent which is responsible for filtering out inappropriate information while being proactive by discovering new information sources. More sophisticated systems would allow the search to continue while the user is not connected to the network or present at a machine. A typical approach would be to send an agent off into the network with a task of gathering information on a topic, disconnecting and waiting for the agent to complete the activity. This approach can allow a user to be more productive while the search is in progress and also save money through not having to be connected for the duration of the search.

2.5.2 Monitoring Remote Resources

Management of remote resources is currently under investigation by a number of hardware centric companies such as Hewlett-Packard. They are interested in sending agents to remote devices in order to monitor status of the device and signal failures to the administrator.

2.5.3 Market places and Auctions

Market places are meeting places for agents concerned with buying or selling something on behalf of their owner. Buying agents are interested in purchasing something on behalf of their owner and selling agents are interested selling something belonging to their owner. The market place facilitates the interaction between agents allowing buying agents announce their wish to purchase something, new selling agents to be introduced to existing buyers and the subsequent negotiation between buying and selling agents in order to strike a deal. The market place defines the language protocol that is used and so long as the agent knows the language, they can participate in conversations. Auctions are different to Markets in that they use formal bidding protocols for price negotiation which all participating agents must be aware of, such as Dutch and Vickrey auction protocols. There are many prototype auction houses currently be researched such as that described in [20]; a Java-based framework for a market bidding protocol.

2.6 The "Personal Assistant" Metaphor

The "personal assistant" (PA) metaphor has arisen from a need to help computer users deal with the increasing complexity of the environments and systems they interact with. They are essentially interfaces which are personalised for a particular user, interact collaboratively with their user and learn or gain competence from their user by watching (or being taught) how the user accomplishes a task. They may also be proactive and can offer (or be told) to take responsibility for certain tasks which may be bringing to the user's attention an email message that should be read immediately or filtering out news messages which will not be of interest to the user.

Much research work has been done in the area interface agents and personal assistants ranging from the basic properties they may possess to virtual reality based visions [23]. PAs may be modelled as agents that present a familiar and personalised interface to their user and assist the user by performing appropriate tasks so freeing the up the user's time. Much of the thrust of the research into PAs has been into intelligent assistants; these learn users habits by watching what they do at their machines through a variety of techniques including user programming, knowledge engineering or machine learning techniques to assist the user in dealing with the various applications they interact with. There are many examples of agents that learn their users preferences such as the electronic mail agent described in [12] to the Microsoft Agent [15] which is essentially framework within which a programmer can create animated characters that react to a user's input.

Having said that much work had be done on interface agents and personal assistants, not much work has been done in the area of network based PAs which is the area that FollowMe will focus on. Within FollowMe, a PA may have to interact with the user via a number of devices so that the user can initiate tasks and receive the results of those tasks in a number of different environments. It must be highly available to the task agents it has dispatched so it can continue its co-ordination however the user may well be disconnected from it for long periods of time. For example, a user travelling in their car could dial up their PA via a hands-free cell-phone and have a personalised selection of news articles read out. This could be viewed as an extension of the Personal Assistant metaphor; instead of the PA residing close to the user at all times, the PA has the ability to migrate into the network when the user is disconnected where it can remain available to the tasks it has launched and be contactable by agents for mediation.

2.7 Agent Languages

The subject of "agents" has probably generated more special purpose languages than any other area of computer science. These range from languages with a *strong* commitment to agent intelligence, particularly with respect to anthropomorphic concepts of belief, desire and intention, to so-called *weaker* languages with a greater emphasis on the software engineering. As the aim of the FollowMe project is not to develop *intelligent* agents, but *mobile* agents, this document will only include discussion of languages which are weak with respect to concepts of agency. For an in depth review of strong agent languages the reader is referred to [24]. While this may appear to be a great loss, the shift to more object based languages means that the real functionality of the agent is captured by the components which make it up, rather than by the programming framework it is expressed in.

The role of the agent description language is to tie together the various components that make up an agent, and to shield the user from details of the underlying mechanisms. Agent languages are scripting languages typically with features (or components) supporting mobility and communication. In the context of *mobile* agents, the agent designer should not be concerned with detailed programming tasks such as the suspension and resumption of threads when an agent is moved from one location to another. While the designer requires the power of a real programming language, we would also want to offer a flexible prototyping system that many compiled languages fail to deliver. This is the realm of so-called scripting languages.

2.7.1 Scripting Languages

Scripting languages can be distinguished from conventional programming languages along a number of different axes; interpretation versus compilation, strong versus weak typing, low-level versus high-level primitives, static versus dynamic allocation, and their support for object-orientation and threading.

Scripting languages do not require a separate compilation phase before execution. The aim is to reduce the time spent going round the traditional development cycle (edit-compile-execute), the script writer need only edit and execute. Although this means scripts are less efficient than their compiled cousins, this is not a problem as they should not be used for time consuming, computer intensive tasks.

Conventional programming languages are increasingly strongly typed, ensuring that data is not lost in accidental type conversions. This places the burden on the programmer to explicitly indicate where valid type conversions can occur. By comparison, scripting languages are generally weakly typed, trading rigour for ease of use. These factors mean that the development time of a scripted application is shorter than with traditional approaches.

With the increasing use of inter- and intra-net based systems, scripting languages have become increasingly important as a means to connecting together different system components. Whereas programming languages like Java are designed to build systems from the ground up, scripting languages generally assume the existence of the necessary high-level objects. The script provides a simple way of connecting these objects together, acting as a kind of "component glue". For example, the Tool Command Language (TCL) is used to arrange the layout of GUI components within a window, and Unix shell scripts assemble a number of different processes into a pipeline. Scripting languages and programming languages are complementary in this respect.

With compiled languages, it is efficient to use statically defined structures wherever possible. This applies to every feature of the language; including data declarations, function or method definitions, function or method declarations, and class definitions. The variables in a scripting language are generally highly dynamic structures. For example, whereas Java provides both fixed and variable length strings, only dynamic strings are found in JavaScript, simplifying the choice of primitive data types. In an object-based scripting language such as JavaScript even the classes themselves are created on the fly. Scripting languages also typically have good support for garbage collection, saving the user from the headaches of discarding unused data and objects. This feature is by no means unique to scripting as Java also provides good support for garbage collection.

Like most other computer systems, scripting languages are becoming increasingly object-based or object-oriented. As noted above, JavaScript provides very basic support for the definition of classes and the creation of objects. The JavaScript language provides no built in class inheritance mechanism, although again much the same effect can be explicitly created by the programmer in the dynamic construction of an object. This is a deliberate omission as

inheritance can be seen as having a negative effect on object encapsulation; exposing the innards of a class to its subclasses. These dependencies ultimately reduce the potential for re-use of script fragments.

The complexity of the threading model for scripts is kept to a minimum. Whereas conventional programming languages like Java provide a sophisticated multi-threaded model, most scripting languages stick to a single thread. Because of the increasing use of scripts to define user-interfaces, as in TCL/TK and JavaScript, the only complication to this model is the ability of scripts to respond to user events. The choice of whether event handlers are best implemented by interrupts or as separate event threads is open to debate. However, as event-handlers should generally be short-lived we should not need to worry about the problems of synchronisation and deadlock which plague concurrent systems.

2.7.2 Scripting Architectures

It is important to distinguish between a scripting language and its underlying scripting architecture. The scripting architecture is a foundation comprising the basic services available to the scripting language. The advantage of this division is that different vendors are free to introduce their own scripting languages providing they are compliant with this architecture. The spirit of interoperability is one of the main objectives of the Object Management Group (OMG), whose proposed CORBA component model RFP (orbos/97-06-12) calls for an architecture that maps directly onto JavaBeans. The main architectural points of interest are event based messaging, with *introspection* as the means of determining an object’s interface.

2.7.3 Scripting the Desktop

Although simple scripting can be seen in the Job Control Languages of early mainframe systems, it is on the desktop that they have gained in popularity. The early Macintosh user interface sported a ground-breaking graphical user interface, yet for years it lacked any principled mapping onto a well-defined language. Scripting languages allow GUI operations to be recorded, and then played back to control interface operations. CORBA has made few inroads onto the desktop, which may be due in part to the fact that CORBA currently provides no scripting language support. The CORBA scripting language RFP (ORBOS RFP9) calls for a language that fits naturally into the proposed CORBA component model. It is likely that either Netscape’s JavaScript or Sun’s Jacl (The language formerly known as TCL), or both, will be conscripted into this role. Microsoft’s domination of the desktop is nearly total; Visual Basic has evolved from its humble beginnings as Microsoft Basic to a flexible and all-encompassing scripting language. As a consequence, the kind of people writing programs is no longer restricted to trained programmers, but now includes a new breed of end-user programmers. Scripting languages are ideally suited to the demands of these new users.

2.7.4 Scripting the Web

Web based distribution is currently based on the client-server model. The norm today is for the server to deliver content in the Hyper-Text Mark-up Language (HTML), or it may provide services via the Common Gateway Interface (CGI); with CGI back-ends increasingly being written in the Perl scripting language. While CORBA is weak on scripting, it offers a comprehensive framework for service interaction.

2.7.5 Scripting for Agents

Agent based computing breaks out of the traditional client-server model, allowing peer-to-peer interaction; *the object web*. Mobile agents demand additional flexibility by allowing processes to jump from one machine to another. With this cross platform capability, script interpretation must be available on every platform the agent may reside at. This may be a native interpreter as in Netscape's current implementation of JavaScript, or the interpreter itself may be cross-platform in that it runs within the Java Virtual Machine (JVM), as with Sun’s Jacl. In the case of mobile agents, it is important that the script interpreter provides support for the suspension and resumption of a script. The saved state, or *continuation*, must be serialisable for transmission. Ideally, this would be transparent to the running script. The only existing scripting languages with agent-friendly implementations are Tcl, TeleScript and ACTOR. Of these, TeleScript was the only commercially oriented product but has so far met with little success, other than contributing the concept of *places* to agent lore. ACTOR is quite an old language now and supports a style of massively concurrent computing designed for parallel rather than distributed architectures. Tcl is a very strong contender in the agent world, especially since its reincarnation as Jacl, the Java Command Language. Despite the non-existence of a specifically

agent friendly version, JavaScript should be ruled out. In terms of user familiarity JavaScript wins through, and since its adoption by the ECMA (www.ecma.ch), JavaScript is now an open standard.

3 Requirements and Analysis

This section reiterates the requirements elicited from [4] and [5], and through analysis produces models that can mapped to a design that is consistent with these requirements. Firstly, a Domain Object Model will be presented, then the roles of the domain objects will be described and lastly, analysis of the requirements will be presented through interaction diagrams over the domain objects. This section closely follows Jacobson's analysis method in [11].

3.1 Domain Object Model

The purpose of the Domain Object Model is to show all the objects present in the WP and over which the use-cases are realised. The domain object model captures the objects present in problem domain through analysis of the use-cases. This analysis was documented in DD2 and the resulting domain object model is presented in figure 1, together with other domain objects the Autonomous Agents objects must interact with. Associations are shown together with notional labelling of the kinds of interaction which will occur. It should be noted that these domain objects are derived from the pilot application requirements. If those requirements change or other requirements become apparent, there may be a need to alter figure 1 to reflect the changes.

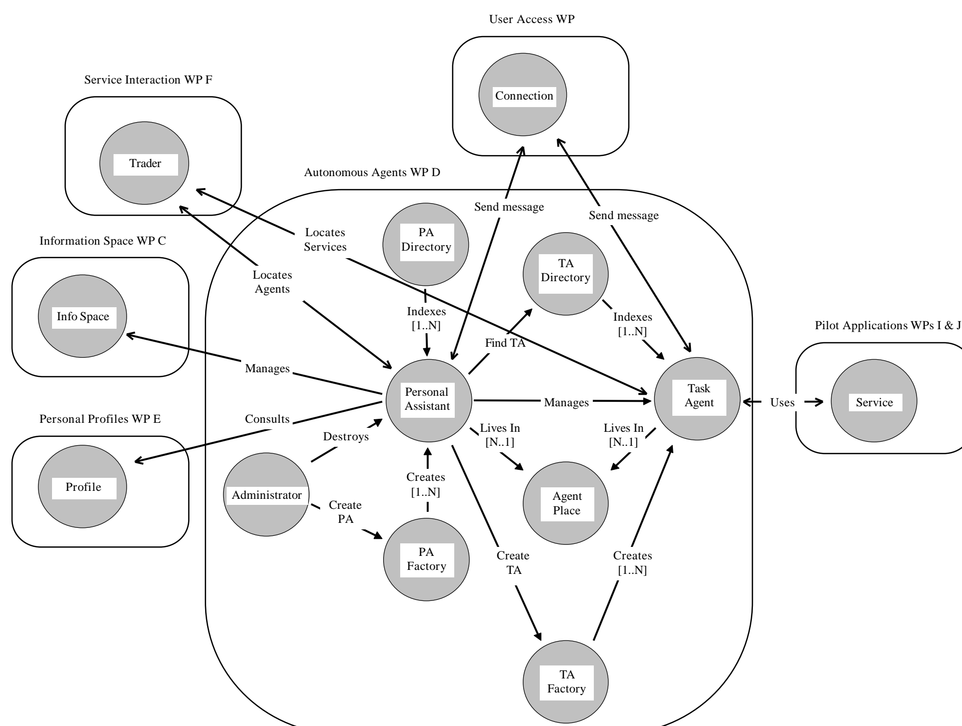


Figure 1- Domain Object Model

3.2 Roles of the Domain Objects

3.2.1 Administrator

The administrator is a role really played by an actor and is included here for completeness. The role could be played a human user or specialised agent; initially we envisage it played by a human user. The administrator will initiate the creation and destruction of Personal Assistants (PAs) and will be responsible for the general maintenance the domain objects below.

3.2.2 Agent Place

The role of an Agent Place is to present an abstraction of the host machine to agents that live in the place. The Agent Place will hide the detail of different host machines and provide a consistent environment in which agents execute. It will provide access to other non-mobile objects it is hosting such as a Trader.

3.2.3 Personal Assistant

The PA plays a mediator. It facilitates the selection, initiation and subsequent tracking of task agents. It will provide a convenient interface to the diary object (see DE3) and will be capable of movement in order to effectively track and relay reports to the user. To simplify the agent framework, only one PA per user will be allowed although this could be extended in the future to allow specialisation of PAs. A typical life-cycle of a PA may include the following:

1. PA is created
2. User connects to PA
3. User requests a list of available services
4. PA gathers and returns information
5. User selects a service to use
6. PA present agents available to interact with service
7. User selects an agent
8. PA initiates creation and launching of agent
9. PA records agent as dispatched
10. User disconnects from PA

3.2.4 Personal Assistant Factory

The role of this object will be to create PAs and it will be a static service provided by each Agent Place. The Agent Place will provide access to the factory.

3.2.5 Personal Assistant Directory

The is a directory service which will allow PA to contacted. PAs may well have to move between Agent Places to ensure continuous operation and this service hides the location of the PA.

3.2.6 Task Agent

Task Agents (TAs) represent their owner when interacting with services and other agents. They carry out specific tasks on behalf of their owner. TAs like PAs, execute in an Agent Place and have the ability to move between Agent Places if required. A typical life-cycle of a TA could be as follows:

1. TA gets created by a Task Agent Factory
2. TA gathers all data necessary to complete its task
3. Dispatches itself to service
4. Uses service and returns results to PA

5 Destroys itself

There are variations possible in previous sequence for example a TA may interact with a service over a long period of time and send several sets of results

3.2.7 Task Agent Factory

The Task Agent Factory will be to create TAs and it will a be static service provided by each Agent Place. The Agent Place will provide access to the factory.

3.2.8 Task Agent Directory

The Task Agent Directory is a service which allows TAs to be contacted. TAs may well have to move between Agent Places and this service hides the location of the TA.

3.3 Analysis of Use-Cases

The following analysis will be presented through typical usage scenarios initiated by the user.

3.3.1 Create a PA

A user downloads a FollowMe distribution package and installs it on their local machine. During the installation, an AgentPlace is created on the user's machine that initially contains no agents. In order for the user to have an agent representative, they need a PA which must be created and a connection to the PA established. This is achieved by the administrator of the place which in this case will be the user, obtaining a reference to the PA Factory from the Agent Place and creating the PA. The PA gathers the details to enable a Personal Profile for the user to be created which include a name for the PA, a password which must be given when the user wishes to open a new session with the PA. The PA is registered with the PA Directory, an Information Space (IS) is created and a Personal Profile created is deposited in the Information Space.

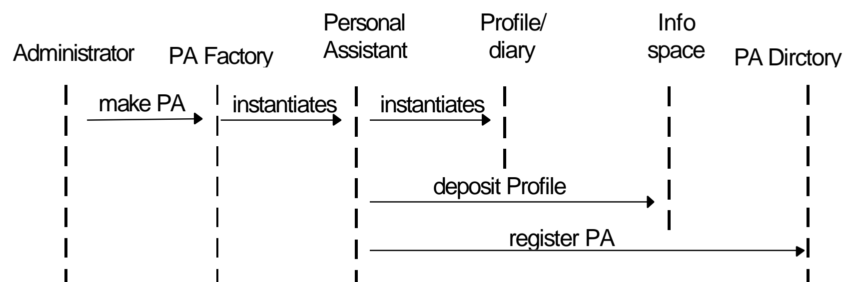


Figure 2 Create personal assistant

3.3.2 Connect to PA

The user connects their machine up to a network and starts a local AgentPlace. A "splash" screen is sent to the users device and the user enters the name of their PA together with their password for validation. If successful, a connection is established to the PA which sends its first screen to the connection. This screen will allow the user to select one of the options: list all the services the available, select a currently executing task, initiate a task, move to local Agent Place and disconnect. These cases will be presented in the following sections.

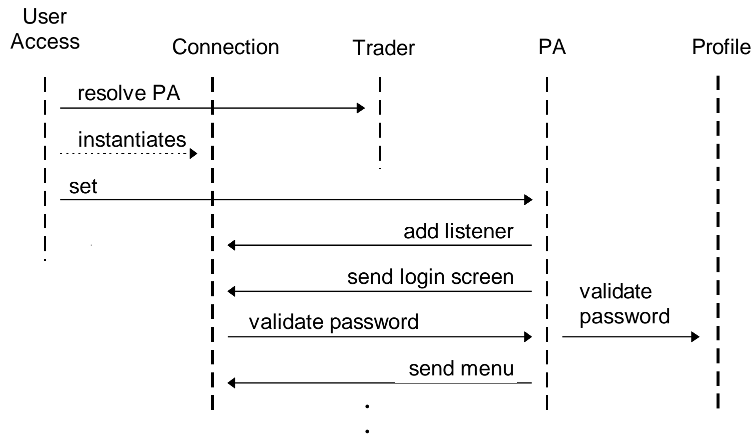


Figure 3 Connect to PA

3.3.3 Disconnect from PA

The user wishes to end the current session with the PA.

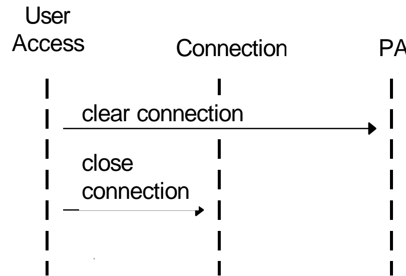


Figure 4 Log-out initiated by user

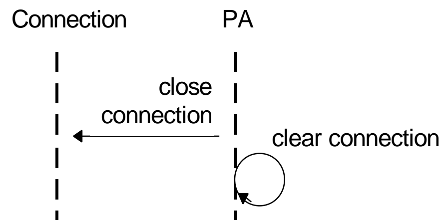


Figure 5 Log-out initiated by PA

3.3.4 List Services

The user wishes to see the services available. The PA contacts the Trader requesting all of the service profiles available.

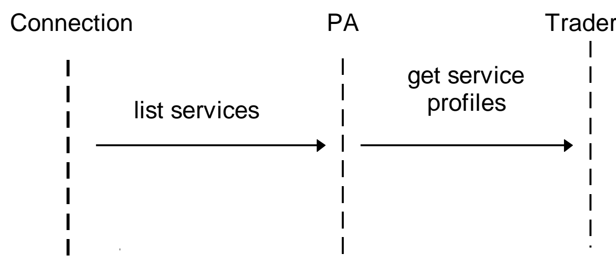


Figure 6 List services

3.3.5 Select a Task

The PA maintains a list of all currently executing tasks. When the user wishes to get the status of a task it must first select that task. The PA must assemble a list of the currently executing tasks for the user to browse and select.

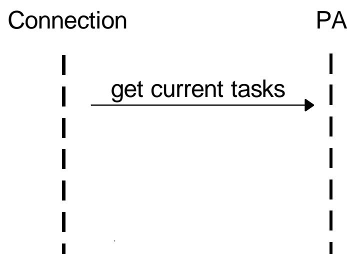


Figure 7 Select a task

3.3.6 Initiate a Task

Once the user has selected a task agent from an available list, they will probably want to launch it. The PA invokes an operation on the TAFactory supplying as a parameter the Agent Profile which the TAFactory uses as a "template" for the new agent. The agent is set executing in the current AgentPlace and may need additional user input in order for its task to be completely defined. Once all required input has been received, the agent follows its mission. The PA records the Task Agent as dispatched. Initiating a task involves the following activities:

- The selection of a service against which the agent is to run.
- The selection of an agent capable of interacting with that service. Agents may be defined in the service profile or maybe selected from a list favourites.
- The creation of the agent.

The interaction diagram in Figure 8 assumes that a connection has been established with the user and that the service has been registered with the trader.

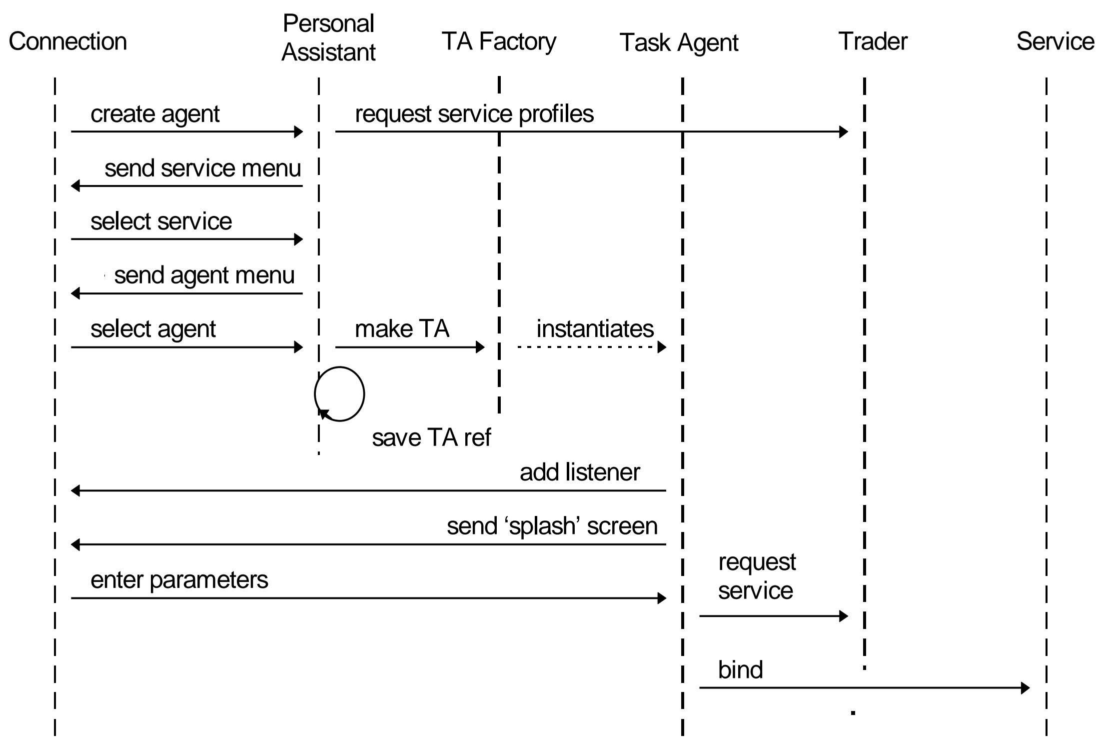


Figure 8 Create task agent

3.3.7 Signal to Agent

One way communication with an agent would, for example, allow a task agent to be cancelled.

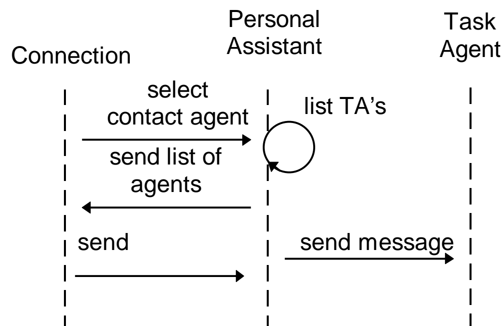


Figure 9 Signal to task agent

3.3.8 Communicate with Agent

Assume that the user is already logged in with a connection. We arrive at this point from the main menu of the Personal Assistant, hence the opening 'select contact agent'.

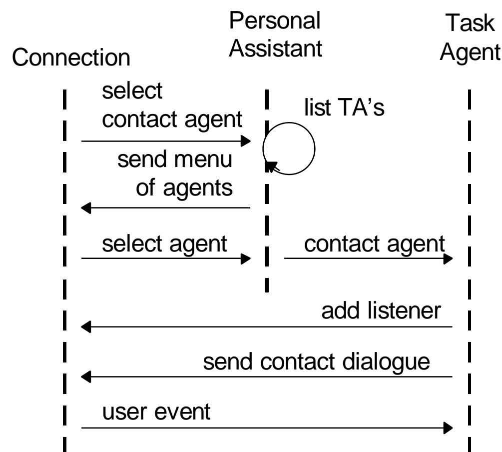


Figure 10 Contact agent

3.3.9 Report to User

The scenario here is that an agent has either finished its task or needs to rely an interim report to the user and it is essentially a communication initiated by the agent.
The simplest scenario is a simple one-way delivery of information, where the user is currently on-line.

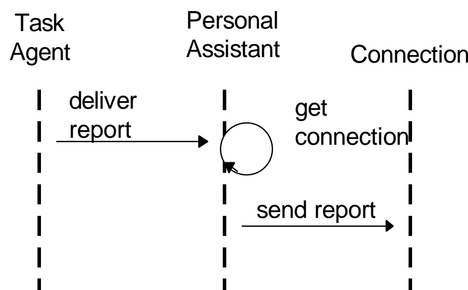


Figure 11 Report to on-line user

The next diagram covers the case where the user is not online and the Personal Assistant takes the initiative in establishing a connection.

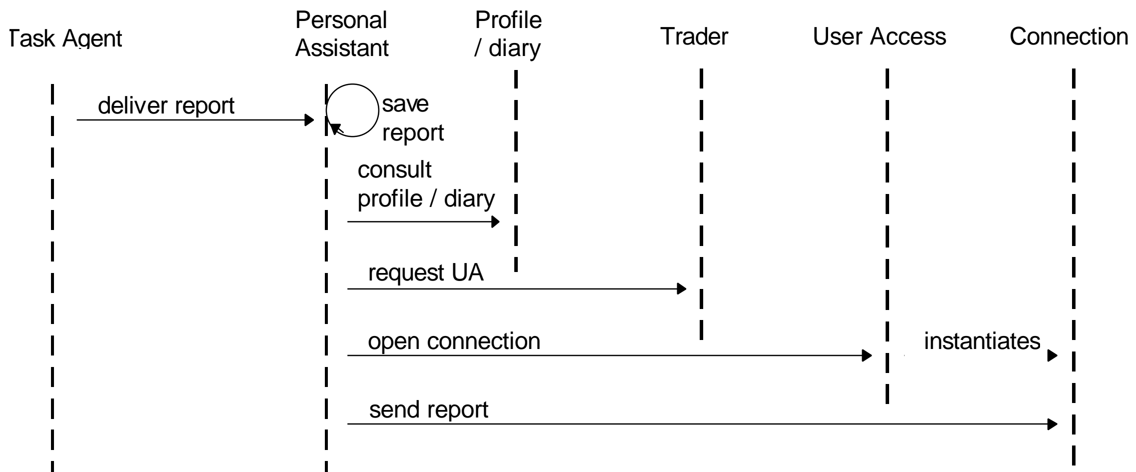


Figure 12 Report to off-line user

3.3.10 Communicate with User

Communication initiated by a task agent requiring the user to supply additional information to enable the completion of the task.

Figure 13 illustrates the case where the agent wishes to contact the user. The Personal Assistant is involved in setting up the connection, but passes control back to the agent once this has been established. The simplest case is where the user is currently on-line.

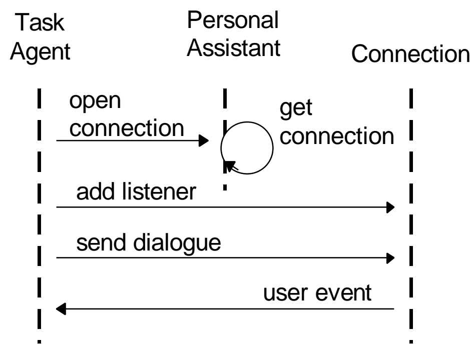


Figure 13 Contact on-line user

Figure 14 assumes the user is not currently on-line but is contactable.

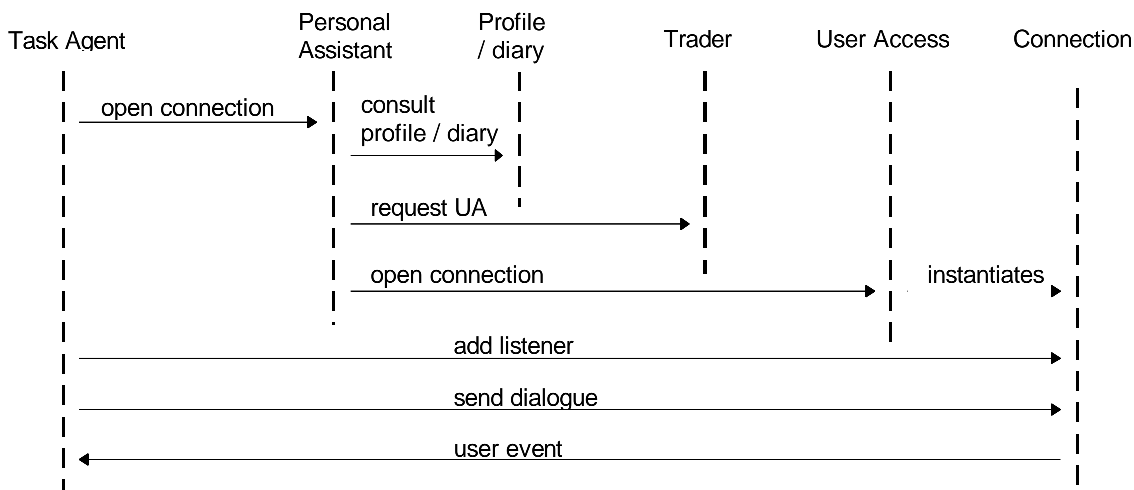


Figure 14 Dialogue - user is off-line but contactable

3.3.11 Request Move to Remote AgentPlace

The AgentPlace in which the PA is residing is about to shutdown, so the PA is informed that it must migrate to another trusted and log-lived AgentPlace where the PA can reside unimpeded. When instructed, the PA must migrate to the “well-known” AgentPlace or one that is user supplied and continue operation. The PA must inform the Trader of its impending move, then migrate. At the new AgentPlace the PA must inform the local Trader of its presence.

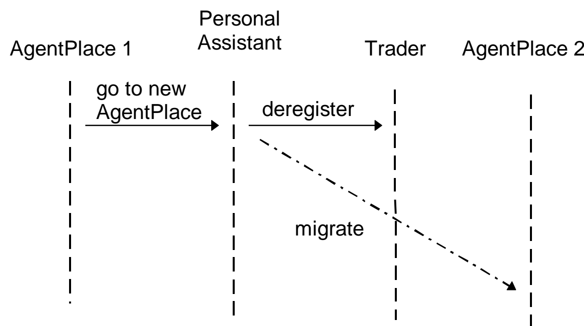


Figure 15 PA migration to remote AgentPlace

3.3.12 Request Move to Local AgentPlace

The gains access to device connected to a network which is capable of running an AgentPlace. An AgentPlace is created and through UserAccess, a request is made to locate the user’s PA. Once a dialogue is established, the user can request that the PA migrate to the local AgentPlace. To accomplish this, the PA must inform the local Trader of its intention to move and then migrate. At the new AgentPlace the PA must inform the local Trader of its presence.

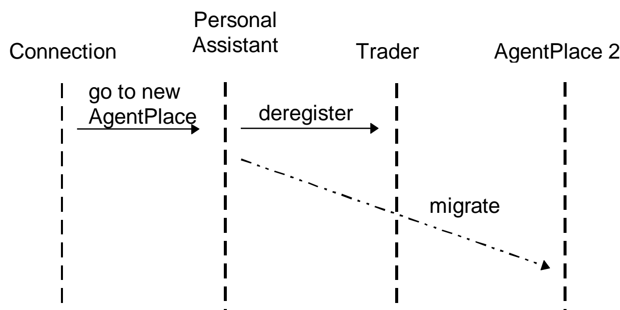


Figure 16 Recall PA from remote AgentPlace

3.3.13 Destroy a PA

When a user wishes to be permanently removed from FollowMe, the user must instruct the Administrator to destroy the PA. The administrator establishes a connection to the PA and invoke a destroy operation on the agent. This will prompt the PA to check whether there are any outstanding tasks; if there is, it will initiate actions to cancel the tasks (or obtain user agreement to do so). Once this is complete, the entry in the Trader is removed, the Information Space and anything in it is removed and the PA is destroyed.

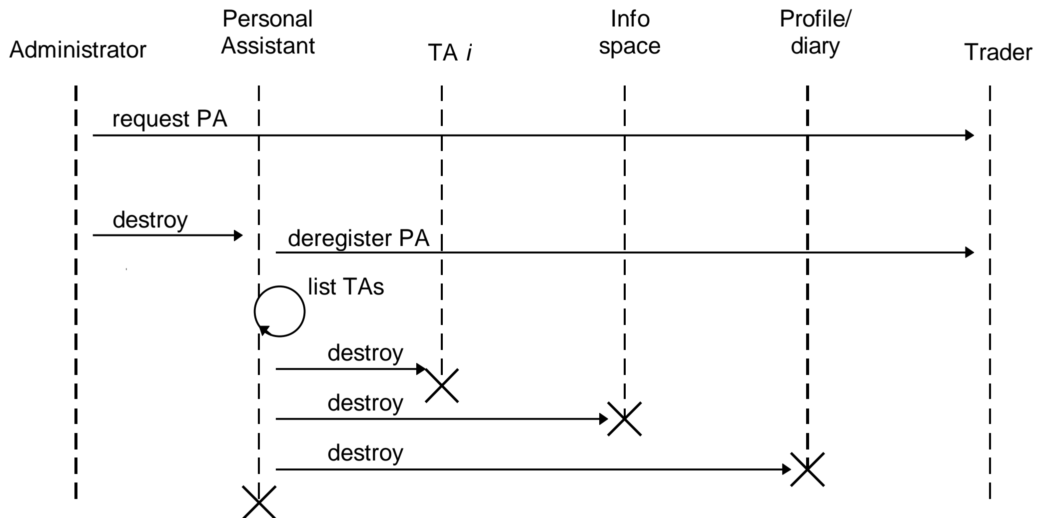


Figure 17 Destroy personal assistant

4 Design

In order to realise the agent framework, the Autonomous Agents WP and indeed the Personal Profiles and Service Interaction WPs will use the facilities and functionality provided by the Mobile Object Workbench (MOW) WP. The MOW provides us with *Mobile Objects* which “live” in a host abstraction known as a *Place*. Mobile Objects can move between Places taking any data they may need with them. The MOW `MobileObject` class will provide the basis for our agents and the `Place` interface and `PlaceImpl` class will provide the basis for AgentPlaces. Additionally, we will use a call-back mechanism provided by the MOW Place implementation to allow the recording of agents arrival and departure. This will be the basis of a rudimentary auditing facility for AgentPlaces which it must be stressed will not be comprehensive and is there to allow us to explore the issues of trust and mistrust in agent systems. As such, this will be one of the research areas we will explore.

4.1 Factories

Factories are a well established pattern documented in [28] and are used to create objects. Autonomous Agents needs to provide facilities to create instances of TAs and PAs. This will be done by PA Factories and TA Factories based on the factory pattern. It is envisaged that the functionality provided by these factory components will be merged so there is only one Factory in an AgentPlace which will be capable of manufacturing PAs and TAs.

4.2 Traders and Directory Services

PA Directory Services and TA Directory Services are required to keep track of PAs and TAs “in the field”. They will enable PA and TA names to be resolved and then communicated with. The Trader is part of the Service Interaction WP however, due to its close association with the Autonomous Agents WP, the framework will be described briefly here. Our intention is to bring all the directory services together and combine them into a AgentPlace Trader, one (and only one) of which will be present in each AgentPlace. The motivation behind this is to reduce the number directory service contact points to one. Trading is based on service type; services export their offer of service via their interface type to the Trader. Associated with the service offer maybe properties of a particular service implementation. Therefore, the Service Interaction WP must derive a type hierarchy and the permissible property-value pairs that may be used with each type. Possible types include Agent, PersonalAssistant, TaskAgent and Profile. Some of these types will have property-value pairs associated with them, for example PersonalAssistant type will have a property called “name” which will be a string and will be unique in the Personal Assistant name-space.

By allowing Traders in AgentPlaces to be federated (co-operate with one another), objects that are not present in the local AgentPlace can be resolved. For example, when a user wishes to contact their PA, they will issue the request to the local Trader which will be able to provide a reference to the PA even if it not present in the local AgentPlace. For more details on Trading, see Service Interaction DF3.

4.3 Agents

In order to construct an agent framework we require agents. Two types of agent will be provided; PersonalAssistant and TaskAgent. Figure 18 shows the inheritance hierarchy of the agents in the framework. The classes `PersonalAssistantImpl` and `TaskAgentImpl` are the default implementations of the interfaces PersonalAssistant and TaskAgent. Class `Agent` is abstract will not be instantiable.

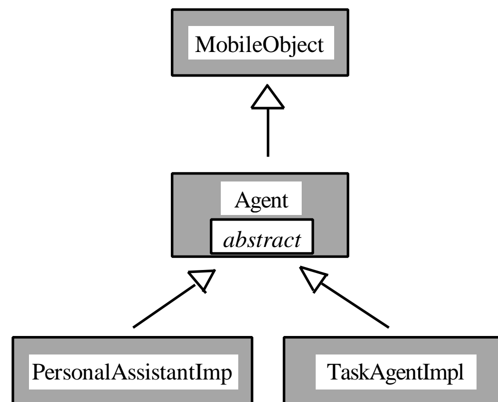


Figure 18 - Hierarchy Diagram

There are mobility requirements for both TAs and PAs; a TA must be able to migrate to a new AgentPlace where a service is located, and a PA must be able to move between AgentPlaces to ensure continuous operation. Mobility is accomplished through use of the MOW facilities. A typical usage would be:

1. Resolve a service type specification with the Trader
2. Move to location of the service

To allow dynamic behaviour and late binding in the pilot applications, and to allow redundancy in service provision, service resolution with the Trader and migration should happen sequentially. This is to try and ensure that the agent will be moving to an active and not a stale service. However, if this situation should arise, the agent must repeat the resolution process and attempt to move to a new service.

Since mobility is required by both types of agent implementations, it make sense to put the mobility facilities in the `Agent` class.

4.4 Task Agents

Central to our definition of a software agent is that it acts autonomously on behalf of the user. One implication of this is that we must take all steps necessary to create an agent framework that is as publicly visible as possible. To increase the base of users able to understand and write their own agents, we took the approach of basing this framework on scripting languages. These are generally a simpler class of programming languages which lack the full sophistication of a conventional programming language, in return for ease of development.

The FollowMe Technical Annex introduces the concept of an agent description language by which the high level behaviour of the agent may be defined. Besides specifying agent behaviour, an equally important function of this description is that it is open to inspection by the user responsible for the agent. The role of agent description language is ideally suited to a scripting language. Whereas conventional programming languages like Java are designed for building systems from the ground up, scripting languages assume that all the main components exist already, and is a highly visible way of 'gluing' these components together.

From the early stages of the project design, the eXtensible Mark-up Language (XML) [27] has been used extensively to represent the personal profile, diary information and the service profile. Within the User Access work-package, a

specialised form of XML, XSL, is being used to define user interface layouts, achieving device independence. The ECMAscript language, derived from Netscape's JavaScript, is currently the most popular scripting language used in conjunction with XML. ECMAScript [3] provides the basis of the XSL scripting language [26]. In addition, the Dynamic HTML (DHTML) Document Object Model [25] will provide a standard object model for both HTML and XML documents. The W3C will supply bindings for Java and ECMAScript [1]. *The task agent language will therefore be an implementation of ECMAScript*. The language definition is not reproduced here, but can be found at the European Computer Manufacturers' Association (ECMA) site [3].

4.4.1 Creating and Using Agents

The normal mechanism by which a FollowMe user instantiates an agent is via their PA. The PA can retrieve either user-defined agents held in their own information space, or agents developed by service providers to go with their own services. In the latter case, these agents are seen as a natural extension of the service description, a meta-object distributed alongside a service offer and held by the Trader (see Service Interaction work-package). These are the two main sources of the agent scripts described in this work-package.

A new TA is constructed around a mission object. A mission defines all the components that go to make up an agent, including profile objects and scripts. Profile objects are defined by XML profile objects embedded directly within the agent mission, and may include personal information or diary data. Profile objects may run independently of the script itself, enabling the diary, for example, to generate internal events within the agent. The mission may be defined by a service provider and supplied to the user as part of the service profile, or enthusiastic users may write their own agent missions.

A script is evaluated by a script object which maintains the name space available to running scripts. The name space includes the mission itself and all of its sub-components. In this way the agent has access to all the major components of the task agent. As the agent binds to new services, these are added to its name space.

4.4.2 Scripting Basics

ECMAScript is derived from the JavaScript language developed by Netscape Communications Corporation to add client-side functionality to web pages. ECMAScript thus draws on a huge user base of both casual and professional programmers. JavaScript in turn adopts many of the control structures found in the Java language, but not its programming model. ECMAScript is an object-based language, but is not object-oriented in the sense that it does not support the concept of class inheritance.

Within the scripting language we need to make a clear distinction between the functionality of the objects embedded within the language, and the keywords and control structures of the language. The former includes basic domain objects such as the personal assistant and the agent place through which most of the agent's work is done. However, it is the scripting language that provides the framework which makes it possible. The objects themselves are arranged into the hierarchy presented in the next section, while this section provides a basic introduction to the language itself. Because all the specific functionality required by agents can be encapsulated within the object hierarchy, it has been possible to keep the scripting language largely as it is defined in the ECMAScript standard (ECMA-262).

ECMAScript provides a few basic control structures which will be familiar to Java (and C) programmers. In the following descriptions, italicised text represents well-formed script, and bold text represents valid ECMAScript keywords and symbols. Code enclosed within square brackets is optional. The ECMAScript syntax is formally defined in appendix A, using Backus-Naur notation.

- The conditional, 'if', statement.
  if (***expression* **) *statement* [**else** *statement*] **;**

- The iterative, 'while', statement.
  while (*statement* **)** *statement* **;**

- The iterative, 'for', statement.
  for (*initialisation* ; *test* ; *increment* **)** *statement* **;**

A variation on the standard ‘for’ loop allows enumeration over the members of an object.
for (*variable* **in** *object* **)** *statement* **;**

- All of the iterative statements above can be terminated early.
  break ;
  Or we can exit just the current iteration.
  continue ;

- A construct familiar to Pascal programmers is the ‘with’ statement. This allows some simplification of statement blocks containing nested object structures. The expression may be automatically prepended to object references where appropriate.
  with (*expression* **)** *statement* **;**

- Declare and optionally initialise one or more variables.
  var *var1* [= *expression1*] **,** *var2* [= *expression2*] **,** … **;**

- In addition, ECMAScript includes `function’ statements, which dynamically create new object member functions (methods).
  function ([*arg1*, *arg2*, …] **) {** *statements* **}**

- Finally, a function may return a value, terminating execution of the function.
  return [*expression*] ;

ECMAScript provides the basic data types listed below, though all variables, formal parameters and function returns are un-typed. As in Java, there is also a primitive **null** value.

- number
- string
- boolean

An ECMAScript implementation should also provide access to the predefined classes, Object and Array.

Wherever a script is used, it must be enclosed between `<script>` and `</script>` tags. These tags demarcate the extent of the script within an XML document.

4.4.3 Object Hierarchy

While scripting is important to our definition of agents, scripts operate in an environment where many of the objects it is using are predefined; the script is merely the 'glue' by which these components are stuck together. Even though the ECMAscript language is essentially the same as that found in a web browser, it is this environment that makes it an agent language rather than a client-side browser language. These components are organised into a hierarchy which defines the context of the running script.

4.4.3.1 Missions

At the top of the hierarchy is a mission object, which encapsulates the agent's functionality and data. Objects immediately below this are accessed as properties of the mission object. The mission represents the name-space available to running scripts, including the mission itself and all of its sub-components. As the agent binds to new services, these are added dynamically to its name-space.

To effect mobility, the agent requires a control object on which it can invoke simple instructions like 'jump' which transports the entire agent to another location. As the complete agent is represented by the mission object, it is this object that provides access to the necessary mobility functionality. Because everything is a part of the mission, a valid short-hand is simply to leave the 'mission' prefix off any reference to a member of the mission, so the 'jump' instruction may be used on its own as in the example above.

To simplify the task of the agent with respect to contacting the user, most of the business of user interaction is handled by the personal assistant (PA). Every agent maintains a remote reference to the PA that launched it, through which it can deliver results for later viewing, or obtain a connection for direct two-way interaction.

The agent also maintains a reference to whatever place it is currently located. Through the place the agent can contact the local trader (and via this the federation of traders) which is the primary access point for all services available to the agent.

Finally, the agent is equipped with its own information space where it can deposit results and other working data. When this space is first created, the personal assistant may provide access to central copies of the personal profile and personal diary.

This object hierarchy, comprising a mission, the personal assistant, a reference to the current place and an information space, along with their respective properties, is shown in Figure 19. The hierarchy is shown in the form of a UML object diagram. See appendix B, and the documentation for related work-packages for the APIs to these individual objects.

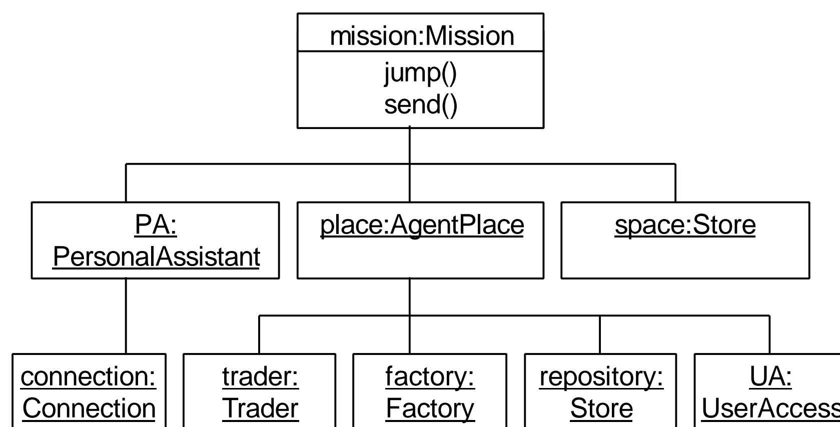


Figure 19 Object Hierarchy

4.4.3.2 Parameters

A mission may additionally include explicitly defined components, described in XML. XML is a structured content description language in that it allows us to describe the information content of an object in terms of significant conceptual entities. XML is not used (in this instance) to define the semantic content of an object. In effect, these components define object literals. Conventionally, only primitive data types such as numbers and booleans may be expressed as program literals. The content mark-up does not define a low-level representation, or serialisation, of an object. Nor indeed does the conceptual structure represented in the mark-up, necessarily correspond to the actual object structure of the implementation, of which it is independent.

These additional components can be thought of as parameters passed to the agent when it is first created. Before any script is executed, these parameters are added to the mission object, alongside the existing object hierarchy. These parameters are defined within the <mission> tag, which has optional name and description attributes (see appendix B for a description of these XML tags). Like all XML, the mission may contain nested elements, each of which maps onto a separate object.

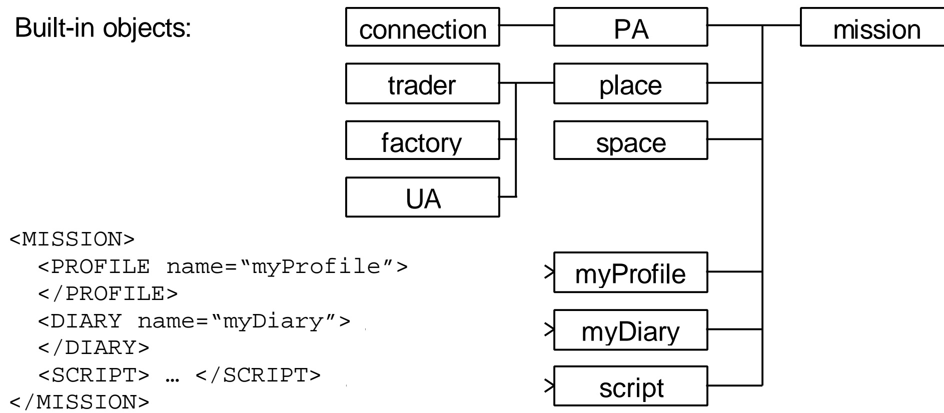


Figure 20 mission with parameters

As stated above, the XML defines only the content of the object, but not any additional functionality we may require, such as data validation. In the case of the personal profile and diary, defined in the personal profiles work-package, this functionality is provided by Java classes. The agent programmer is given the option of associating a given XML tag with a Java class, using an XML processing instruction. In different circumstances, the same XML element may be implemented by different classes. These association rules should be thought of as part of the mission style, which is strictly separate from its content, but they need not occupy a separate style document. The following processing instruction associates a fully qualified class name with an XML tag, and should appear before the mission itself.

```
<?RULE UK.ac.uwe.ics.followMe.profile.Profile implements PROFILE ?>
```

The mission may now contain a description of a profile which will be used to construct a new Profile object which is added to the mission.

```
<MISSION>
  <PROFILE>
    <ID>Steve Battle</ID>
    <EMAIL>sab@ics.uwe.ac.uk</EMAIL>
  </PROFILE>
</MISSION>
```

If the mission encounters an object for which no implementation has been defined, then it constructs a generic information object as defined in the personal profiles design.

The storage of mission components need not be uniform. The mission may maintain references to other mobile objects. The choice of whether to place an object in an information space or to incorporate it into the agent itself depends on whether it represents an information object or an active object. Active objects which depend upon having their own thread cannot be held in the information space, while large passive information objects can be cut free from the agent making it lighter and easier to move.

The mission may contain a wide range of different objects, many of which may become active at different stages of the agent life-time. The mission must cater not only for profile objects which are passive entities, or information objects, but also for diary objects (see the personal profiles work-package) which are continually active and responsible for the sourcing of local timing events.

The mission also contains the scripts which define the agent functionality. Where many scripts reside in the same mission, each contained within their own `<SCRIPT>` and `</SCRIPT>` tags, the mission policy is to pass control from one to the next sequentially.

4.4.3.3 Dynamic properties

New objects can be created at any time during the lifetime of an agent, and these enter the scope as new members of the mission. In the following script, a new variable, x, is added dynamically to the mission, and we can refer to it as 'mission.x'. As before, we have the option of dropping the 'mission' prefix.

```
<SCRIPT>
  mission.x = 1 ;
  y = x ;
</SCRIPT>
```

Instead of classes defining the structure of objects statically, new objects may be created dynamically. Imagine that we need to create a pizza object, with a given base and with extra toppings. ECMAScript provides generic Object and Array classes to which we can add new properties as we need them. The following script constructs a cheese & tomato 'pizza' with extra anchovies and olives.

```
<SCRIPT>
  pizza = new Object() ;
  pizza.base = "cheese & tomato" ;
  pizza.toppings = new Array("anchovies", "olives") ;
</SCRIPT>
```

4.4.4 User Interaction

Interaction with the user is modelled on the idea of a server which fields requests for text pages, albeit pages containing XML mark-up. The mission object acts as a user interaction adapter, translating page requests into accesses against mission properties and functions. When a user wishes to get in touch with an agent, they are aided by the personal assistant which places an initial request for a 'contact' page. To satisfy this request, the agent mission need only contain an appropriately named object which is extracted and 'stringified' for transmission to the user.

Interaction with the user is modelled on the idea of a web server which fields requests for text pages. Instead of HTML being the primary mark-up language, XML is used as a content description language, which can be transformed into a viewable form by the application of XSL style rules. This transformation occurs within the user access module, but the content and style must be delivered on request. Interaction with an agent may also be seen as a stateful interaction, extending over a number of pages delivered sequentially. Two useful entry points into this hyper-text have been described as the *splash-screen* and the *contact screen*. The splash screen is the first screen sent from the agent to the user when it is first created. At this point the agent may gather additional data it requires for its mission. The contact screen is an interface the agent should define for when the user wishes to get in touch with the agent. At this interface the user may trigger some new activity, or even destroy the agent.

Because user access is an external service, the details of its interface are defined outside the scope of this document (see User Access work-package). The agent is shielded from the complexities of user interaction by the personal assistant which manages the user connection and its immediate event handling. The personal assistant maps the functionality of this interface onto the scripting model. Within this document, the `send()` method is used on the mission to communicate with the user. This method can be used to send simple (XML) strings, or complete pages to the user.

The requests from the user access module represent user events. Some of these events may be requests for more information such as style sheets or linked pages as described above, while others may require further processing by user-defined event handlers. Where this kind of event handling is required, the event is first routed back to the object which represents the page sent to the user. Within the scope of the FollowMe project, this object would correspond closely to an HTML forms object, with one or more submit buttons. When the user clicks on a submit button, the contents of the form would be returned to the sender along with the identity of the activated submit button. The form details are used first to populate the form properties for use in later processing. Then the form object would locate the button element associated with the event and look for its *onClick* attribute, which may refer to a user-defined event handler; defined simply as an ECMAScript function. For an example of this process, see the Pizza Factory example later.

4.4.5 Script-directed Mobility

To transport an agent from one place to another involves first suspending execution of the script, 'freezing' its current state; that state is then copied to the destination and then resumed. Most of the problems associated with mobility arise from the difficulty of explicitly manipulating Java language control threads. An object cannot be transported from one

place to another until all control threads involved with that object have been shut down. The underlying mobile object workbench provides low-level support for shutting down threads in preparation for a move, and for creating a new thread once the object has arrived at its destination. This programming model makes strong demands of the mobile workbench programmer. To perform this kind of thread suspension directly in Java would require that the designer explicitly check to see if the suspension command has been issued. To make a script interruptible at short notice like this, places strong constraints on the design of the script interpreter. It must have minimal dependence on the state of the execution stack represented by its own Java thread, because this thread state is non-serialisable. The script interpreter must adopt what is known as a continuation semantics, where whatever it must perform next finds explicit representation as a continuation structure.

The scripting language provides an isolation layer, hiding the agent programmer from the sophistication of the mobile object workbench. In the scripting model, the suspension and resumption of control is provided invisibly. The programmer is free to include jumps from one place to another within code blocks such as loops and function bodies (rather than purely at the end of code blocks is in the MOW). This would be useful for example, within an explicitly represented travel itinerary which the agent could step along within a loop.

```
<SCRIPT>
  for (i=0 ; i<itinerary.length ; i++) { jump(itinerary[i]) ; ... }
</SCRIPT>
```

4.4.6 Threading

As well as simplifying mobility, the scripting model also aims to simplify the threading model for agents. By avoiding complicated multi-threading models we can avoid the use of complex concurrency control constructs such as synchronisation and locking [19]. When an agent is created, it is provided with a single thread of control. From that point on there is no way for the agent to create another thread other than creating an entirely new agent. The model encourages a one-to-one association between threads and agents.

This is not to say however, that other threads cannot pass through the agent concurrently with the main thread. It is desirable for the agent to be able to respond to external messages, which include messages from other agents and services, and to user interface events. These are all threads that are temporarily 'borrowed' from their sources. In the case of user interaction, the interface would appear to lock-up if the agent decided to hijack the event thread for its own ends. To keep the interface reactive, the relevant event-handlers should be short lived.

The main requirement for inter-agent (and service/agent) interaction is that they are able to invoke methods on each other. Besides providing the internal environment for running scripts, the mission also defines the agent's interface to the outside world. All properties of the mission are deemed private, so that nobody can access the agent's parameters or variables without consent (using defined get and set functions). Any functions that are defined within a mission script are potentially exposed to the outside world, however the default visibility on all functions is private, so there is no risk to the forgetful programmer.

To make a function publicly available, the script element it is defined within should have its visibility attribute set to 'public'. This is one use for breaking down the mission into separate script elements, one for the public interfaces, and the other containing private functionality. Note; there is no 'protected' visibility mode because this determines visibility under inheritance and is therefore an object-oriented, not an object-based feature.

4.4.7 The Java connection

Many of the objects available to a running script are actually Java objects, so the script object must be capable of accessing the properties of, and invoking methods on these objects. A Java object can be accessed dynamically in this way, only if it supports Java *reflection*.

Because ECMAScript is weakly typed and Java is strongly typed, the actual Java method that is selected for invocation may depend upon the actual values of the parameters. Automatic type conversion is used extensively for information passed between the two programming models.

ECMAScript also has access to the Java package hierarchy, and consequently to Java Class definitions. Using this facility, a script may invoke class methods or instantiate new Java objects.

4.4.8 The Pizza Factory

The following example describes a Pizza Factory with a create your own pizza service. Because the interaction is stateful, the `PizzaFactory' creates a service handler called `order', a basic pizza base to which we can add extra toppings. Such a mission might be defined within the service profile (see Service Interaction) of an Internet 'pizza' company.

The agent mission contains two components. The first is a "PizzaMenu" form which can be thought of as the 'splash' screen of the agent. This form lets the user select a pizza base and check various extra toppings as desired. Because this is an information object it can be placed in an information space. To see what actually happens we need to look at the second mission component which is a script.

```
<mission name = "PizzaFactory" >
  <form name = "pizzaMenu">
    Pizza base :
    <select name = "base">
      <option>cheese & tomato</option>
      <option>vegetarian</option>
      <option>ham & mushroom</option>
    </select>

    Extra toppings :
    Anchovies <input name="topping" type="checkbox" value="anchovies"/>
    Olives    <input name="topping" type="checkbox" value="olives"/>
    Cheese    <input name="topping" type="checkbox" value="cheese"/>

    <input type="submit" name="OK" value="OK" onClick="ok()"/>
    <input type="submit" name="cancel" value="cancel"/>
  </form>

  <script visibility="public">
    function ok() {
      jump(service) ;
      order = service.orderPizza(base.value) ;
      for (i=0 ; i<topping.length ; i++)
        if (topping[i].checked) order.extra(topping[i].value) ;
      pizza = order.finished() ;
      send("pizza") ;
    }

    service = place.trader.resolve("PizzaFactory",null) ;
    if (service!=null) send(PizzaMenu) ;
</script>
</mission>
```

If we follow the script we can see that the first thing it does is to define a function called ok(). This function is added to the mission and because it is defined within a public script, it may be used externally. We will return to this function when it is used.

The first real activity the agent performs is to attempt to resolve the named "PizzaFactory" service against the local trader. If this were to fail there would be no point in the user filling out the form. If the result of this is a non-null service reference the form is sent to the user via the Personal Assistant.

We will assume at first that the user has entered their details and pressed the 'OK' button. The PA intercepts and recognises the submission event. It then routes the event to the form which looks up and invokes the corresponding ok() function back on the originating script.

The effect of this is for the agent to jump to the place which hosts the PizzaFactory service. When it arrives it proceeds to order a new pizza. The 'order' object referred to here is a service handler, dedicated to creating this pizza alone. After working through the extra toppings, it completes the order which returns the finished pizza object. The finished pizza is sent back to the user.

If the user were to press ‘cancel’, the form would return straight away as no user defined event handler is specified. The agent would then terminate as there are no pending activities.

4.4.9 Migration Paths

By adopting ECMAScript as the agent description language we provide a migration path from standard web based applications based on HTML and JavaScript to agent applications using XML and ECMAScript. The main elements of the PizzaFactory example can be converted into this framework; the Pizza service itself could be wrapped as a Java Applet embedded in the HTML page and accessed from the script exactly as above.

This approach also gives application writers and service providers within the FollowMe project a means to develop pilot applications before all the major components of the scripting model are finalised. The results of this background work would be useful in their own right as standard Internet services, and as points of comparison with the final agent based services.

Looking at agents from this point of view also highlights the differences between standard web-based and agent-based approaches.

- With the standard approach, the functionality (script) is tied to the page. There is no scope for client side scripts to assemble information from many information sources.
- With the standard approach the execution is tied to the browser. Our approach leaves it up to the script to decide where it is best run; for either efficiency or security reasons.
- Because they are not tied to the browser, agents may work in the background and contact the user wherever they are. The user is not tied to the desktop where they started the agent.

4.4.10 Goal-oriented Description

Goal oriented descriptions permit a still higher level view of agent behaviour than the script alone. The technical annex calls for mechanisms to describe, select and adapt agents based on the user's goals. These goal-based mechanisms are not critical to the development of the main pilot applications and are consequently more research based than developmental.

We have approached this problem from the point of view of the ends the user is trying to achieve, based on the service model. The service model comprises the entities and relations defined by a particular service contract. A given agent script may contain assertions (using the <assert> tag) expressed in the same service model (see Service Interaction design, DF3). These assertions should reflect the actual conditions established by the script. Agents may therefore be selected on the basis of these assertions.

4.5 Personal Assistants

A PA is a special agent which is personalised for the use of a specific user. There will be one PA per user. The role of the PA is to launch and destroy TA, manage TA which have been previously launched, manage the reporting requirements of the TAs, and manage the users Information Space. It also contains the `ProfileInterface` object (see DE3) which allows the manipulation of the user’s Profile.

The implementation of the PA will consist of two control objects which will manage all the administrative activities of the PA and will be described in the following sections.

PAManager

This object will perform the general functions of the PA such as managing the connections between User Access and the PA, resolution of services and agents, compilation of lists of services and agents for presentation to user. It will implement the methods destroy, getConnection and setConnection. It will also implement the means by which currently executing TAs are kept track of in the form of a list.

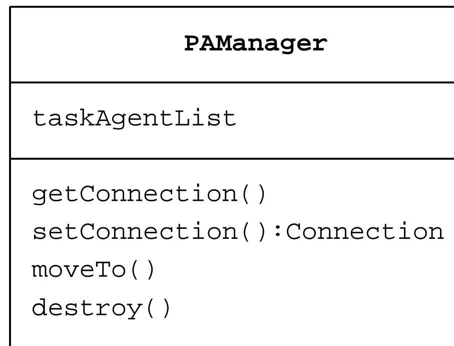


Figure 21 PAManager class diagram

ReportManager

This will manage the reporting requirements of both the user and the TAs. For example, when a TA deposits a data item into the user's Information Space, it will inform the ReportManager that it has done so by calling the `deliver` method on the PA which is implemented by the ReportManager. This allows the ReportManager to evaluate the connectivity status of the user and decide whether to deliver the data item or to defer through consultation of the diary.

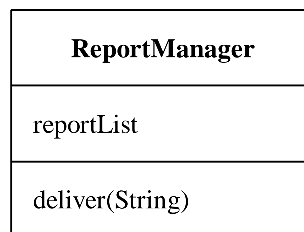


Figure 22 ReportManager class diagram

4.5.1 Movement of PAs

When a user has to shut their machine down, they will also shut the AgentPlace down which may well be hosting the user's PA or other agents. In order for the PA to keep on managing ongoing tasks, it must be available on the network. One way of achieving this is for the AgentPlace to warn all the agents it is hosting to move because the current place will shortly cease to exist. The PA will then jump to a well-known (or user-specified) AgentPlace where the PA can reside long-term and monitor its tasks and relay reports.

When the user connects their machine back up to a network, establishes a local AgentPlace and becomes contactable via a network, one of the first things they will do is establish a connection to their PA. Once this is achieved, they can request their PA to move to their local place. This extends the Personal Assistant metaphor to include not only organisational abilities but also locality of interaction, much like a secretary that travels with the user. In the electronic context, this is useful since long lived exchanges between the user and PA may be conducted locally reducing network overheads.

4.5.2 Uses of the PA

The personal assistant is the main point of contact for the user and, through user access, will guide the user through a number of activities involved in the creation and management of agents.
When the user first contacts the personal assistant, they are presented with a menu of the activities available to them. These include the following.

- select agent
- select service
- contact agent

- collect results

By selecting an agent, the user can instantiate a new agent based on any mission saved in the user's own information space. This may be an agent written by the user, or adapted by the user from an agent originally supplied by a service provider, tailoring it to their own needs.

Other sources of agents are the service profiles defined by the service providers. The service profile (see the service interaction work-package) is a meta-description, registered with a trader, of the service, including both the interface definitions and sample agents which operate against those interfaces.

If the user wishes to contact active agents, PA constructs a list of all the agents it has launched and are still running. On selecting one of these, control is passed to the 'contact' page of that agent. The details of this are specific to that agent, though it should at least offer the option to terminate the agent. When the contact dialogue is complete, control over the interaction is passed back to the personal assistant.

Agents may deliver results back to the PA for later viewing by the user. The objects to be collected are held in the information space. When the user logs in, the personal assistant should notify the user whenever results have arrived.

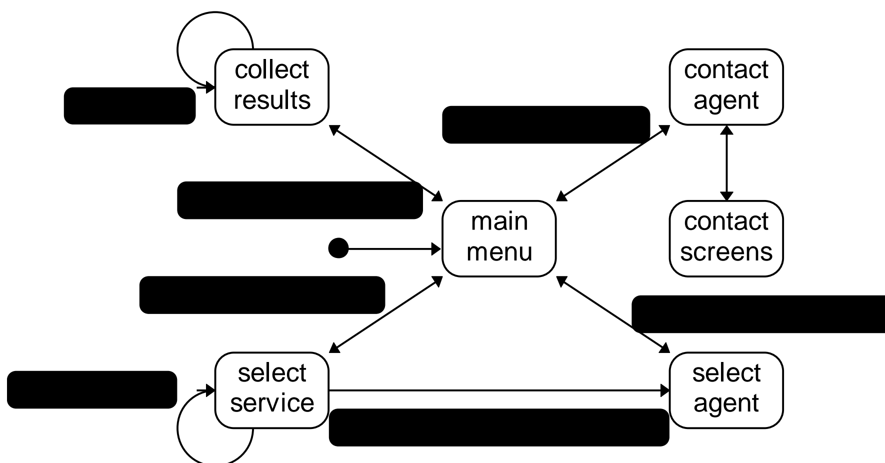


Figure 23 Personal Assistant state-diagram

4.6 AgentPlaces

The MOW provides us with a Place within which a MobileObject may execute. We will extend Place to AgentPlace which will be the host abstraction within which our agents will execute.

AgentPlace will integrate the AgentFactory as previously mentioned, Trader service and the means by which the Place will be administered; the AgentPlaceImpl object. An AgentPlace will also integrate UserAccess. These services will all be based on the MobileObject class and will present their interfaces in the same manner as mobile objects however they will not be mobile. We envisage a boot-strapping process which starts all the necessary services when an AgentPlace is started.

4.6.1 Auditing Features

As was mentioned earlier, the ability for AgentPlaces to keep a log of the agents it is hosting and has hosted is not explicitly required by the pilot applications however, if there is to be any industrial strength deployment of an application using the FollowMe framework, then auditing facilities will be needed to ensure that agents are well behaved and to enforce non-repudiation. In this first implementation of audit trails, AgentPlaces will keep a log agents that have visited the AgentPlace.

The following events will be logged through internal method invocations and will record information such as the agent's name, where it came from, where it is going, time spent in the AgentPlace:

Agent Arrived
Called when an agent arrives at an AgentPlace

Agent Created
Called when a new agent is created

Agent Dispatched
Called when an agent leaves an AgentPlace

Agent Disposed
Called when an agent is destroyed

4.6.2 Starting an AgentPlace

We will initially provide an AgentPlace as a Java application that is run from a command line or console window. Future developments may allow an AgentPlace to the run inside a browser however this is not currently available from the MOW.

When an AgentPlace is started, services which provide specific functionality have to be started also. As a first implementation an AgentPlace through its constructor will instantiate an Information Space, a Trader, a User Access service and a Factory. References to these services will be obtainable through the interface to the AgentPlace. An AuditLog will be created and through the call-backs provided by the MOW, methods will be implemented that are called when any of the events described in section 4.6.1 occur. These methods will update the log with details of the event and time-stamps.

4.6.3 Implementing an AgentPlace

AgentPlace will be specified as an interface and will have a default implementation of `AgentPlaceImpl`. A class diagram of `AgentPlaceImpl` is shown in figure 23.

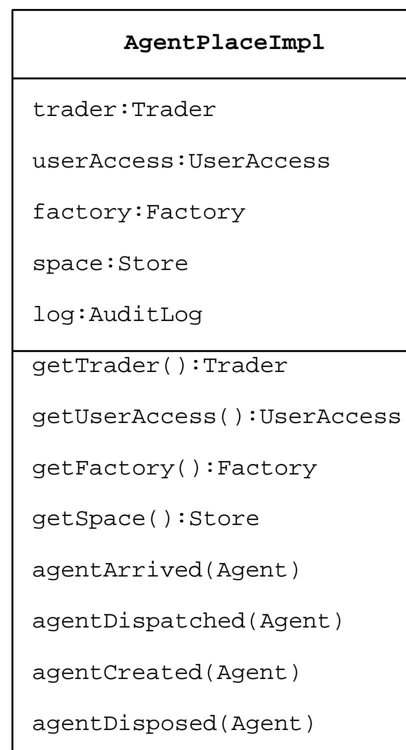
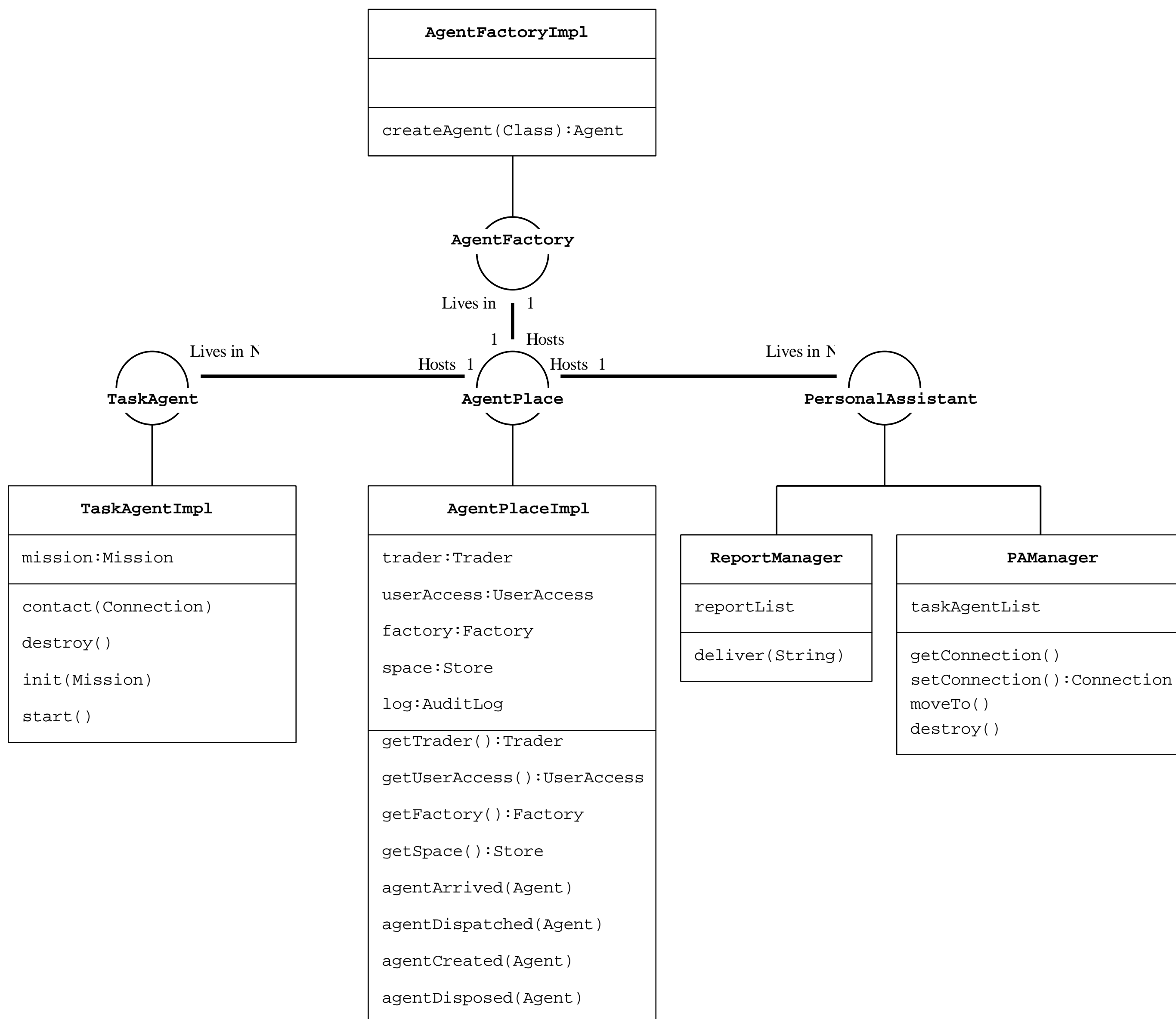


Figure 24 AgentPlaceImpl class diagram

4.7 High-level Class Diagram



4.8 Summary

In this chapter we have described a design for the Autonomous Agents WP which when combined with the Service Interaction and Personal Profiles WPs, will form the Agent Framework. In particular, this WP addresses the mobility of users and through the PA, will allow users to initiate tasks that may be completed while they are off-line.

5 References

[1] BYTE, Weaving a Better Web, <http://www.byte.com/art/9803/sec5/sec5.htm>, March 1998.

[2] Chavez and Maes, Kasbah: An Agent Marketplace for Buying and Selling Goods, Proceedings of the 1st International Conference on the Practical Application of Intelligent Agent and Multi-agent Technology, pp 75-90

[3] ECMA, Standard ECMA-262, ECMAScript Language Specification, <http://ECMA.ch/>, June 1997.

[4] FollowMe, Requirements of FAST Pilot Applications, FollowMe deliverable DI2

[5] FollowMe, Requirements of INRIA/TCM Pilot Applications, FollowMe deliverable DJ2

[6] Franklin and Graesser, Is it an agent, or just a Program?: A Taxonomy for Autonomous Agents, 3rd International Workshop on Agent Theories, Architectures and Languages, Springer-Verlag, 1997, pp 21-35

[7] General Magic, Odyssey, <http://www.genmagic.com/>

[8] General Magic, Telescript Language Reference, <http://science.gmu.edu/~mchacko/Telescript/docs/telescript.html>.

[9] IBM, Aglets Work Bench, <http://www.trl.ibm.co.jp/aglets/>

[10] IEEE Computing, IC Online Virtual Roundtable, <http://computer.org/internet/online/v1n4/round.htm>, The Future of Software Agents

[11] Jacobson, Object-oriented software engineering: A use case driven approach 1992

[12] Lashkari, Metral and Maes, Collaborative Interface Agents, Proceeding of AAAI 1994

[13] Maes, Agents That Reduce Work and Information Overload, Communications of the ACM, Vol 37, No 7, pp 31-40

[14] Mitsubishi, Concordia, <http://www.meitca.com/HSL/Projects/Concordia/Welcome.html>

[15] Microsoft Agent, <http://www.microsoft.com/intdev/agent>

[16] MOLE, <http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole.html>

[17] ObjectSpace, Voyager, <http://www.objectspace.com/>

[18] OMG, Mobile Agent Facility, <http://www.genmagic.com/agents/MAF>

[19] John Ousterhout, "Why Threads Are a Bad Idea (for most purposes)", <http://www.scriptics.com/people/john.ousterhout>, 1995.

[20] Rodriguez et al, FM96.6: A Java-based Electronic Auction House, Proceedings of the 2nd International Conference on the Practical Application of Intelligent Agent and Multi-agent Technology, pp 207-224

[21] UMBC, AgentWeb, <http:/www.cs.umbc.edu/agents>

[22] White, Mobile Agents, Software Agents, editor Bradshaw, pp 437-472, MIT Press, ISBN 0-262-52234-9

[23] Wloka and Green, The Virtual Tricorder, Brown University, Dept of Computer Science, 1995, CS-95-05

[24] Wooldridge and Jennings, Software Agents, IEE Review, Jan 1996, pp 17-20

[25] W3C, Document Object Model, <http://www.w3.org/TR/WD-DOM/>, March 1998.

[26] W3C, A Proposal for XSL, <http://www.w3.org/TR/NOTE-XSL.html>, August 1997.

[27] W3C, Extensible Mark-up Language, <http://www.w3.org/TR/1998/REC-xml-19980210>, February 1998.

[28] Gamma et al, Software Design Patterns

Appendix A - BNF for ECMAscript

```
Literal ::= NumberLiteral | StringLiteral | BooleanLiteral | NullLiteral

NumberLiteral ::= <NUMBER_LITERAL>

StringLiteral ::= <STRING_LITERAL>

BooleanLiteral ::= "true" | "false"

NullLiteral ::= "null"

Name ::= <IDENTIFIER>

VariableDeclarator ::= Name ( "=" Expression )?

Expression ::= Assignment | ConditionalExpression

Assignment ::= PrimaryExpression AssignmentOperator Expression

AssignmentOperator ::= "=" | "+=" | "-=" | "*=" | "/=" | "%=" |
                       "<<=" | ">>=" | ">>>=" | "&=" | "|=" | "^="

ConditionalExpression ::= ConditionalOrExpression ( "?" Expression ":"
ConditionalExpression )?

ConditionalOrExpression ::= ConditionalAndExpression ( "||"
ConditionalAndExpression )*

ConditionalAndExpression ::= InclusiveOrExpression ("&&" InclusiveOrExpression)*

InclusiveOrExpression ::= ExclusiveOrExpression ( "|" ExclusiveOrExpression )*

ExclusiveOrExpression ::= AndExpression ( "^" AndExpression )*

AndExpression ::= EqualityExpression ( "&" EqualityExpression )*

EqualityExpression ::= RelationalExpression ( ( "==" | “!=” )
RelationalExpression )*

RelationalExpression ::= ShiftExpression ( ( "<" | ">" | "<=" | ">=" )
ShiftExpression )*

ShiftExpression ::= AdditiveExpression ( ( "<<" | ">>" | ">>>" )
AdditiveExpression )*

AdditiveExpression ::= MultiplicativeExpression ( ( "+" | "-" )
MultiplicativeExpression )*
```

```
MultiplicativeExpression ::= UnaryNumericExpression ( ( "*" | "/" | "%" )
UnaryNumericExpression )*

UnaryNumericExpression ::=
     ( "+" | "-" ) UnaryNumericExpression | UnaryExpression | PrefixExpression

UnaryExpression ::= ( "~" | "!" ) UnaryExpression | PostfixExpression

PrefixExpression ::= ("++" | "--") PrimaryExpression

PostfixExpression ::= PrimaryExpression ( "++" | "--" )?

PrimaryExpression ::= PrimaryPrefix ( PrimarySuffix )*

PrimaryPrefix ::= Literal | Name | "this" | "(" Expression ")" |
AllocationExpression

PrimarySuffix ::= "[" Expression "]" | "." Name | Arguments

Arguments ::= "(" ( Expression ( "," Expression )* )? ")"

AllocationExpression ::= "new" Name Arguments

Statement ::=
     EmptyStatement |
     StatementExpression ";" |
     IfStatement ";" |
     WhileStatement ";" |
     ForStatement ";" |
     ForInStatement ";" |
     BreakStatement ";" |
     ContinueStatement ";" |
     ReturnStatement ";" |
     WithStatement ";" |
     FunctionStatement ";"

Block ::= "{" ( BlockStatement )* "}"

BlockStatement ::= LocalVariableDeclaration ";" | Statement

LocalVariableDeclaration ::= "var" VariableDeclarator ( "," VariableDeclarator)*

EmptyStatement ::= ";"

StatementExpression ::= PrefixExpression | PostfixExpression | Assignment

IfStatement ::= "if" "(" Expression ")" Statement ( "else" Statement )?

WhileStatement ::= "while" "(" Expression ")" Statement

ForStatement ::= "for" "(" ( ForInit )? ";" ( Expression )? ";" ( Expression )?
")" Statement

ForInit ::= LocalVariableDeclaration | Expression

ForInStatement ::= "for" "(" ForInInit "in" Expression ")" Statement

ForInInit ::= PrimaryExpression | "var" VariableDeclarator

BreakStatement ::= "break"

ContinueStatement ::= "continue"

ReturnStatement ::= "return" ( Expression )?
```

```
WithStatement ::= "with" "(" Expression ")" Statement

FunctionStatement ::= "function" Name FormalParameters Block

FormalParameters ::= "(" ( Name ( "," Name )* )? ")"

Parse ::= ( BlockStatement )*
```

Appendix B - Mission and scripting tags

This appendix defines the mission tag and other typical XML tags that may appear within it. This section covers script tags and form elements. Tags required for personal profiles and service interaction are defined in the corresponding work-packages.

The W3C proposal for XSL defines a set of core flow objects which include basic HTML form elements. This is a good starting point as it allows easy migration from standard web-browsers and HTML content creation tools. XML forms are comprised of text input boxes, radio buttons, check-boxes, menus and click-able images. The interface to User Access provides a means for programs to connect to a particular device and send an XML forms specification of the interface. While the form is in use User Access may post events back to the originating program.

The *<mission>* tag

Functions: defines a service view as an agent
Attributes: name, desc
Contains: <script> and additional XML mark-up

The desc attribute is a short description of the agent mission used by the PA to describe its function to the user.

In addition to its functional script, the agent may contain other XML mark-up, which defines a set of objects available to the agent while it is running. To support user interaction, the agent must carry with it, declarations of the data to be presented or requested from the user, and definitions of the way this data is actually presented. This latter function is to be performed using XSL.

The option of additional mark-up also permits the use of un-scripted agents which can be specified with the use of custom XML. Taking advantage of the extensibility of XML, the developer may wish to use something like the <applet> tag in HTML and associate this with a special purpose loader. This kind of functionality will be available in the first release of the service profile object.

The *<script>* Tag

Functions: Defines an agent script using the ECMAScript language
Attributes: name, desc, visibility
Contains: <script> and additional XML mark-up

The visibility attribute is set to 'PRIVATE' by default. If set to 'PUBLIC' any functions declared within the script are externally available. Event handlers must be declared 'PUBLIC'.

The *<form>* Tag

Function: Defines a form.
Attributes: name
Contains: A form may include other mark-up.

An XML form can appear anywhere within an XML document. It contains the form elements defined below. A form may contain other mark-up elements (such as XSL) especially to label input elements.

The *<input>* Tag

Function: An input element
Attributes: checked, maxLength, name, size, src, type, value, onClick
Contains: nothing

The input tag can be used to define text fields, menu's, and buttons.

text field <input type=text>

A single line text box. The *size* parameter is the suggested width of the box in characters. If this *size* exceeds the limitations of the display device, a smaller text box size can be accommodated with sideways scrolling. The *maxLength* attribute is the maximum number of characters that can be entered. The default values for *size* and *maxLength* are device dependent. An initial default value may be specified with the *value* attribute. If the user resets the form, the field is set to this default *value*.

password field <input type=password>

Identical to text field except that the displayed characters are obscured in some device dependent manner.

checkbox <input type=checkbox>

A checkbox allows users to select and deselect a given option. Each checkbox specifies a single option. The optional *checked* attribute, if declared, indicates that the checkbox should be selected by default. Checkboxes can be arranged into a group by giving them the same *name*. Each option can be associated with a *value*.

Radio Button <input type=radio>

Radio buttons can be arranged into a group by giving them the same *name*. Within any group a single element should be *checked* as the default selection. If no elements are checked, the first element of the group is assumed to be the default. If more than one element is checked, the first checked element is assumed to be the default. Each option should be associated with a *value*.

Submit button <input type=submit>

The value attribute specifies a label for the button. If no value is provided the default value "Submit". The button should have a name. A form may include more than one submit button distinguished by name and value. On submission, an event is generated containing the *name and value* of the button.

Reset Button <input type=reset>

A reset button lets the user clear the form, setting all the values back to their defaults. The default *value* of a reset button is "Reset", but a different label can be specified with the value attribute. A reset button should have a *name*.

Hidden fields <input type=hidden>

Hidden fields are like text fields except that they are never displayed.

File dialogue <input type=file>

Displays a file selection dialogue with the facility to browse an information space.

The <textarea> Tag

Function: Multi-line text input element
Attributes: cols, name, rows, wrap, onChange, onFocus, onBlur
Contains: plain text

Allows multi-line text entry. *rows* and *cols* specify the suggested size of the input box, but this may vary according to device limitations. Plain text between the begin and end tags is taken as the default value of the text box. By default, the cursor only moves to the next line when the user enters return. With *wrap=virtual*, the text is wrapped to the next line as it is entered, but only entered carriage returns are actually stored. With *wrap=physical*, the line breaks inserted by wrapping are physically stored. The default is *wrap=off*. The text-area element may have a *name*.

The *<select>* Tag

Function: Single/Multiple choice menu
Attributes: multiple, name, size
Contains: select content

The select tag can be used to create pull-down menus. The select contains a number of options which form the displayed elements of the menu. If *multiple* is declared then more than one element may be selected at a time; an element is deselected by choosing it again. Normal menu behaviour is the default, allowing only a single option to be selected. The *size* attribute suggests the number of options which should be visible, although the actual displayed format is ultimately device dependent. The menu may have a *name*.

The *<option>* Tag

Function: Define an option for select content
Attributes: selected, value
Contains: plain text

A menu option. Plain text between the begin and end tags specify the displayed option. An option may be pre-selected by declaring the *selected* attribute. The option may have a value different from the text.

Appendix C - Interface Specifications

Interface UK.ac.uwe.ics.followMe.AgentFactory

public interface **AgentFactory**

The AgentFactory interface allows the client to create new instances of Agent. The abstract Agent class is sub-classed by PersonalAssistant and TaskAgent.

Method Index

makeAgent(Class)
Makes an agent of the specified class.

Methods

- **makeAgent**

public abstract Agent makeAgent(Class agentClass)

Makes an agent of the specified class. The agent class must be defined with a null constructor. Parameters (such as the mission) are set directly on the agent itself.

Parameters:
class - The class must be a concrete subclass of Agent.
Returns:
an autonomous agent.

Interface UK.ac.uwe.ics.followMe.AgentPlace

public interface **AgentPlace**

This interface provides access to an agent place. In particular, it defines the interface to the place currently occupied by the agent, and accessible to the agent as ‘place’.

Method Index

getSpace()
This method returns a reference to an information space managed by the agent place as an object repository which can be used in conjunction with the trader.
getTrader()
This method returns a reference to the local trader.
getUserAccess()
This method returns an interface to the local UserAccess module.
getFactory()
This method returns a reference to the local object Factory.

Methods

- **getTrader**

public abstract Trader getTrader()

This method returns a reference to the local trader (if any).

Returns:
Trader as defined in the Service Interaction work-package.

- **getSpace**

public abstract Store getSpace()

This method returns a reference to an information space managed by the agent place as an object repository which can be used in conjunction with the trader.

Returns:
a Store as defined in the Information Space work-package.

- **getFactory**

public abstract AgentFactory getFactory()

This method returns a reference to the local AgentFactory.

Returns:
an AgentFactory reference..

- **getUserAccess**

public abstract UserAccess getUserAccess()

This method returns an interface to the local UserAccess module.

Returns:
UserAccess as defined in the User Access work-package.

Interface UK.ac.uwe.ics.followMe.Mission

public interface **Mission**

This interface exposes only the functionality of the Mission as required from within an agent script. The Mission implementation defines additional methods for mission construction, which are normally used by the personal assistant.

Method Index

jump(AgentPlace)

send(Object)

Methods

- **jump**

public abstract void jump(AgentPlace destination)

This method, invoked by an agent on itself, allows it to jump from one agent place to another.

- **send**

public abstract void send(Object object)

This method allows the agent to return objects back to the user. The mission may in turn invoke deliver on the PA.

Interface UK.ac.uwe.ics.followMe.PersonalAssistant

public interface **PersonalAssistant**

The Personal Assistant interface includes the points of access required by User Access, Task Agents, and the PA administrative interface. Much of the functionality of the PA is handled by an internal event handler conforming to the interface defined in the User Access work-package.

Method Index

deliver(String)
Informs the PA about content held in the information space to be delivered to the user.
destroy()
Terminate the Personal Assistant.
getConnection()
Returns the connection set in setConnection if not null, otherwise the PA attempts to locate the user and open a new connection.

moveTo(AgentPlace)
Instruct the PA to move to a new location.
setConnection(Connection)
The PA will subsequently register an event listener on this connection.

Methods

- **setConnection**

public abstract void setConnection(Connection connection)

The PA will subsequently register an event listener on this connection. The connection can be cleared by setting it to null. Normally called by User Access.

Parameters:
connection - User Access connection.

- **getConnection**

public abstract Connection **getConnection**()

Returns the connection set in setConnection if not null, otherwise the PA attempts to locate the user and open a new connection. Normally called by a Task Agent.

Returns:
User Access connection.

- **deliver**

public abstract void deliver(String name)

Informs the PA about content held in the information space to be delivered to the user. Normally called by a Task Agent.

Parameters:
name - The name of an object in information space.

- **moveTo**

public abstract void moveTo(AgentPlace place)

Instruct the PA to move to a new location. This method is normally called on the PA by itself in response to a user request.

Parameters:
place - An Autonomous Agent place

- **destroy**

public abstract void destroy()

Terminate the Personal Assistant. Normally called from an administrative interface.

Interface UK.ac.uwe.ics.followMe.TaskAgent

public interface **TaskAgent**

This is the main interface to a task agent. This interface is normally hidden from the end-user/agent programmer. Once an agent has been created in the AgentFactory, a mission is constructed from the XML mission description and passed to the agent for initialisation.

Method Index

contact(Connection)
This method provides a way for the user to contact an agent.
destroy()
Terminate a task agent Normally called only by the Personal Assistant
init(Mission)
Initialise a generic task agent by attaching a new mission.
start()
Called after initialisation - a request to start the agent.

Methods

- **init**

public abstract void init(Mission mission)

Initialise a generic task agent by attaching a new mission.

Parameters:
mission - The Mission object is created by the caller

- **start**

public abstract void start()

Called after initialisation - a request to start the agent. The calling thread must be returned, so the implementation of start() will generate the main agent thread.

- **contact**

public abstract void contact(Connection connection)

This method provides a way for the user to contact an agent. The connection is established via the personal assistant which is responsible for sending the contact message. Once contacted, the agent sends an initial 'splash' screen and registers an event listener with the connection. for details of this process, see User Access.

Parameters:
connection - The User Access connection

- **destroy**

public abstract void destroy()

Terminate a task agent Normally called only by the Personal Assistant