ESPRIT Project No. 25 338

Work packages D, E & F

Autonomous Agents, Personal Profiles and Service Interaction

Agent Framework Guide

ID:	Framework Guide v1.4	Date:	15/3/99
Author(s):	Steve Battle, Nick Taylor, John Tidmus, Michael Yearworth	Status:	Draft
Reviewer(s):		Distribution:	Internal Use Only

Change History

Document Code	Change Description	Author	Date
v1.0	Baseline version incorporating Script Guide v1.01	SAB, JPT	11/9/98
v1.01	Reorganised with additional material	JPT	29/9/98
v1.02	Modifications for AgentFramework v1.2	JPT, MY	5/10/98
V1.03	Changes resulting from Internal Review	NPT	11/11/98
V1.1	Modifications for AgentFramework v1.3	JPT, SAB	10/12/98
V1.11	Additional material	JPT	18/12/98
V1.2	Additional material	NPT,SAB	8/1/99
V1.4	Modifications for AgentFramework v1.4	SAB	15/3/99
V1.5	Changes resulting from internal review	JPT, NPT	31/3/99

1 INTRODUCTION 5

2 FRAMEWORK INTEGRATION & CONFIGURATION 6

2.1 Introduction 6

2.2 Configuring the components 6
Scenario 1 - Using GUI Font-end 6
Scenario 2 - Core Framework 7

3 AUTONOMOUS AGENTS 9

3.1 Introduction 9

3.2 Personal Assistants 10

3.3 Task Agents 10

3.4 AgentPlaces 10

3.5 Missions 11

3.6 Scripts 11
The programming model 11
Running simple scripts 12
Differences from the ECMAScript standard 12
New semantics 13
New syntax 13
Mobility 14

3.7 Generic XML Support 14
Attributes 14
Elements 14

3.8 XML Documents 15
You've gotta have style 15
Get Connected 16
HyperLinks 16
Do you submit? 16
Communicating with the user 17
We have contact 18

4 PERSONAL PROFILES AND DIARIES 19

4.1 Introduction 19

4.2 Personal Profiles 19

4.3 Diary 20
Alarms 20
Alarm attributes 20
Format of time definition 21
Alarm recurrence 21

5 SERVICE INTERACTION 22

5.1 Introduction 22

5.2 Service Interfaces 23

5.3 The Trader 23
Trading within the Agent Framework 24
The `TraderAdmin` Interface 24
The Trader Interface 25
Running the Trader 27

5.4 Service Profiles 27

5.5 Service patterns 1: Stateless services 28
IDL meets XML 28
well-formed XML 29
validity 29

5.6 Service patterns 2: Call-Back 31

6 REFERENCES 33

7 APPENDICES 34

7.1 Appendix A - BNF for ECMAScript 34

7.2 Appendix B - Scripted Example 35

7.3 Appendix C - Mission Example 36

7.4 Appendix D - Alarm DTD 37

7.5 Appendix E - Alarm Example 38

7.6 Appendix F - XML Document Example 39

7.7 Appendix G - Interaction diagrams 42
Create a PA 42
Contact a PA 42
Browse Trader 43
Browse iSpace 43
Browse Active Tasks 43
Create a Task Agent 44
Contact Task Agent 44
Report to User 44
Dialogue with User 45
Move PA to Remote AgentPlace 45
Move PA to Local AgentPlace 45
Kill a PA 46

7.8 Appendix H - "hello world" service and client. 46

7.9 Appendix I - DTD for service profiles 47

7.10 Appendix J – Client of stateless service 48

7.11 Appendix K - Type checking against a DTD 49

1 Introduction

The purpose of this document is to provide information to assist users with the Agent Framework available as part of work packages D, E and F. This document is applicable to release 1.4 of the Agent Framework.

Version 1.4 of the Agent Framework includes components that may be used together to enable many generic activities a mobile (or static) agent based system may require. The components include the Trader, Personal Assistant (PA), Task Agent (TA), Personal Profile, Diary (with Alarm facilities), the script interpreter, Agent Place (including PA Factory and TA Factory). For full details refer to “ReleaseNotes.txt” in the Framework install directory.

The Agent Framework aims to provide a set of components, which can be customised for use by a particular application. The central component facilitating this is the Trader, which allows services to “advertise” themselves to third parties. These services can be discovered and used by any client. However, the Agent Framework allows Task Agents to be defined, created and set running on behalf of a user through the use of novel concepts such as “missions”, which provide a way of embedding scripts and other objects within the eXtensible Markup Language (XML). Also, Task Agents can be defined through use of a script language based on ECMAScript [1], with extensions for handling external Java classes and objects borrowed from the earlier JavaScript. The script provides a simple way for users to connect together pre-defined objects and services, customising them to their own ends. Task agents may move between Agent Places which provide the agents with a uniform set of resources, Agent Place discovery is through the Trader.

In the following chapters, the components are described in greater detail and with reference to examples where appropriate.

2 Framework Integration & Configuration

2.1 Introduction

As stated earlier, the Agent Framework is set of components that may be used to create applications which are agent based and potentially mobile. In the following sections, the configuration of the components will be discussed. Since we are not trying to prescribe a pattern of use for the agent framework, two potential scenarios are presented: the first using our own graphical user interfaces and the second without the interfaces. Applications wishing to present a different user interface to their users can, simply by writing their own which make calls on the interfaces provided by the components. Full details of the interfaces provided by the components is available in the documentation accompanying the Agent Framework release.

2.2 Configuring the components

Note: in order to for certain components in the AgentFramework to work correctly, the changes outlined in **Docs/MOWChanges/readme.txt** must be implemented.

This section describes how to get all components included in the AgentFramework to work together. We will consider two scenarios: for a developer using the supplied GUI elements to control the framework, and as a separate case, without the graphical front-end.

Scenario 1 - Using GUI Font-end

The functionality provided by this release allows services to register themselves with the Trader and add associated missions with the services exported to the trader. These services can be browsed via the graphical interface provided by the PersonalAssistant which allows missions to be viewed and launched.

- **Start the TrivTrader:**

```
java -Dflexinet.trader=(<trader address>) UK.co.ansa.flexinet.trivtrader.service.TrivTrader
```

- **Start the UWE Trader:**

```
java -Dflexinet.trader=(<trader address >) UK.ac.uwe.ics.followme.trader.TraderImpl
```

- **Initialise the UWE Trader.**

This class is demonstration code showing how to create contexts in the trader, and load missions. This example creates certain contexts, then loads all missions stored as XML files in context "/Missions". This requires the setting of a system property "install.directory" to point at the base of the Framework installation.

```
java -Dflexinet.trader=(<trader address >) -Dinstall.directory=<install directory>
UK.ac.uwe.ics.followme.trader.test.TraderAdminTest
```

- **Start a number of AgentPlaces**

Start a number of AgentPlaces with different names. The information space location in the filesystem may be specified using the "ispace.location" property (default is to be created in the current directory).

```
java -Dflexinet.trader=(<trader address >) –Dispace.location= <ispace location>
UK.ac.uwe.ics.followme.agentplace.viewer.AgentPlaceServer <name> <locale>
```

- **Create Personal Assistant**

Each AgentPlace has an administrative interface (currently an AWT component) that enables Personal Assistants to be created. Press the Create PA button and a details screen should appear. Enter a name for the PA (all other fields may be ignored) and press OK. If the PA name is unique the PA will be created.

- **View Personal Assistant**

The PA GUI allows a user to graphically view their PA. The viewer provides a graphical view of the Trader, the Information Space and active tasks. Task agents may be launched by double-clicking on the appropriate mission.

```
java -Dflexinet.trader=(<trader address >) UK.ac.uwe.ics.followme.agent.viewer.PAViewer <PA
name>
```

- **Test PA mobility**

If you started more than one AgentPlace, the PersonalAssistant's mobility may be observed by pressing Jump. This instructs the PA to query the Trader for a random AgentPlace other than itself which it then attempts to move to.

- **Add new mission to trader**

If you have created a new XML mission, it can be loaded into the trader using the command line utility AddMission. This takes arguments of the specific context (which must exist), name and description of the mission, and a filename

```
java -Dflexinet.trader=(<trader address >) UK.ac.uwe.ics.followme.trader.test.AddMission
<context> <name> <description> <filename>
```

eg.

```
java -Dflexinet.trader=(127.0.0.1:12345)(0) UK.ac.uwe.ics.followme.trader.test.AddMission
"/ICSC" "PizzaCollector" "Hunts for required pizza" pizza2.xml
```

- **Launch this new TA**

Once the trader is loaded with a new mission, it may be browsed on the PA GUI. Double clicking on a mission and creates and launches an instantiation of the PizzaCollector TaskAgent.

Scenario 2 - Core Framework

In this scenario, no GUI is required, or the user interaction is provided by a different method. This provides a trivial example how custom classes could create PAs and TAs.

- **Start the TrivTrader:**

```
java -Dflexinet.trader=(<trader address >) UK.co.ansa.flexinet.trivtrader.service.TrivTrader
```

- **Start the UWE Trader:**

```
java -Dflexinet.trader=(<trader address >) UK.ac.uwe.ics.followme.trader.TraderImpl
```

- **Initialise the UWE Trader.**

Use custom classes to create required context profiles, offers and missions. Demonstration code in UK.ac.uwe.ics.followme.trader.test.TraderAdminTest provides an example as to how this may be achieved.

- **Start a number of AgentPlaces**

Start a number of AgentPlaces with different names. The information space location in the filesystem may be specified using the "ispace.location" property (default is to be created in the current directory).

```
java -Dflexinet.trader=(<trader address >) –Dispace.location= <ispace location>
UK.ac.uwe.ics.followme.agentplace.AgentPlaceServer <name> <locale>
```

- **Create Personal Assistants**

The running AgentPlaces may be resolved within the Trader (in context "/AgentPlace"). This provides a reference to the AgentPlace interface and enables a custom class to obtain the relevant PA Factory (by invoking **getPAFactory** on the AgentPlace) then to create a Personal Assistant (method **create**).

- **Create Task Agent**

A PersonalAssistant can create a TA after obtaining its MissionDef from the Trader. Custom code can invoke **createTA** on the PA and hence launch the TA created with the MissionDef pattern.

3 Autonomous Agents

3.1 Introduction

One of the primary motivations behind the Autonomous Agents work-package is mobility; Mobility of users, mobility of programs, and mobility of data. With everything on the move, how do you tie everything together? The user's first contact with the agent framework is the personal assistant, or PA. If you are on the move, disconnected from the network, your PA will pick up your messages and keep you informed. Your PA is hub that all your remotely running agents keep in contact with. Your PA keeps track of your information so that you or your agents can access this data on demand. Your PA is the mission-control that lets you create, monitor and sometimes kill your own agents.

Of the software delivered as part of this core package, the PA is primarily a programmatic object that can be customised to fit in with any particular application. However, for general purpose use the PA may be viewed with a stand-alone user interface based on the latest Java Swing technology. This interface offers you three views of the system.

- Location transparency for information objects is provided by the information spaces work-package. The PA lets you browse your information space, so you may, for example, access reports sent back to you by agents in the field. Your information space is also the repository for your own personal profile and personal diary, your 'filofax' in the agent world.

- You may access services and agents designed to use them through the Trader. The PA lets you explore the available services and the contexts they reside in as though they were directories on a normal file-system. New task agents may be launched; simply by double-clicking on the required mission.

- The PA monitors any agents that you have previously launched, and presents this information to the user as an activity list. Simple house-keeping is possible; you may kill agents selectively, or you may even talk to running agents by clicking on their name, which brings up their 'contact' window.

The definition of what an agent does is encapsulated within the idea of a 'mission'. The mission declares all the objects and classes required by the mission. The format for missions is the eXtensible Markup Language, XML that has been adopted across FollowMe. Looking somewhat like HTML, the mission contains a number of mission components, each describing the content of a single object required by the mission. Some of these components may be passive data objects, while others may define active behaviour.

Complex agents may be custom-built by a particular service provider to suit their own services, in which case Java may be the most appropriate implementation language for these active objects. However, part of the goal of the agent framework is to empower end-users to write their own agents for their own purposes. Pre-packaged agents are good for the casual user, but more experienced users will demand more flexibility. The aim was to provide a simple programming interface that hides the underlying complexity, but none of the functionality, of the underlying mobile object workbench. Our solution is to allow agents to be programmed using the JavaScript language, which is accessible to a wide audience of end-user programmers. JavaScript commands may be written directly into the mission, and with no need to compile they may be run and tested instantly. This vision of agency makes the user the main focus of the agent framework rather than the service provider.

The agent framework provides a range of software components which can be included as part of a mission. The most important of these components are the XML support classes. These provide the script-writer with full access to any XML text, including the mission itself, as though it were an object (a Document Object Model), and Document objects which may be used to build simple user interfaces (such as the agent ‘contact screen’) that work with the user-access work-package. With a powerful set of components, the agent programmer may regard the JavaScript as little more than ‘component glue’. This DIY approach to agent construction allows users to build agents as simply as a child might build a lego toy.

The Agent Framework allows “agents” to be defined, created and launched. It introduces the notions of Task Agents (TAs) and Personal Assistants (PAs) which are both agents, but perform different roles as described next. Both TAs and PAs are built upon the MOW and are both fully mobile. Agents execute in and move between AgentPlaces that provide a uniform view to the Agents much like the Java Virtual Machine does for applets. The following sections briefly describe TAs, PAs, and AgentPlaces then go on to describe the way by which agents may be defined. Full details of the interfaces presented by TAs, PAs and AgentPlaces are in the *javadoc* accompanying the AgentFramework release. For further details of the interactions between PAs, TAs and AgentPlaces, see Appendix 7.7.

3.2 Personal Assistants

The PA is a representative of its user/owner. Currently, the framework allows one PA per user and this PA can be created through the Agent Place interface by a utility application. The role of the PA is to enable task agents to be created and launched and to subsequently provide a guaranteed contact point for running TAs. Once these TAs have been launched, the PA monitors them until it is informed that a TA has finished it’s mission and should be terminated in which case, the PA removes the TA from it’s active-task list and destroys the TA. The PA is also responsible for receiving results from task agents and relays them to the user based on whether they are connected or not. The Personal Assistant is mobile and can move between Agent Places to allow continuous operation.

PAs are created in an AgentPlace. Using the GUI-based AgentPlace implementation, a PA may be created by pressing the “Create PA” button (alternatively the corresponding interface operation may be invoked) to which a unique PA name must be supplied. The PA is then created and set running in the AgentPlace with it’s information space created.

3.3 Task Agents

Task agents execute missions. When they are created, they are initialised with a reference to their parent PA and a mission that is then started and the TA starts running. At any time, a TA may send data or a form to the PA which may or may not be delivered to the user depending on their connectivity status. When the TA has finished it’s mission, it informs the PA which then destroys it.

3.4 AgentPlaces

AgentPlaces are hosts to agents and provide a uniform interface to all agents. AgentPlaces encapsulate two factories: a PA Factory and a TA Factory, which allows for the creation of PAs and TAs (the AgentPlace interface provides an operation allowing a PA to be created). The GUI-based AgentPlace implementation provides a view of the agent currently being hosted and this is updated as agents are created, leave, arrive and are destroyed in the AgentPlace. The GUI version also allows agents to be destroyed by highlighting the agent and pressing the “Kill” button. A “Shutdown” button is also provided for a clean shutdown of the AgentPlace.

When creating AgentPlaces, they must be supplied with a unique name since they are subsequently registered with the Trader and can be resolved based on their name.

3.5 Missions

The Mission provides a flexible way for users to compose task agents from a number of smaller components. These components are declared using XML. The Mission loader implements these objects using implementation rules, which associate specific XML tags with java classes.

The easiest way to use these classes is to specify the appropriate implementation rules and let the mission loader build them for you from the XML you supply as part of the mission. However, all these classes may be used to instantiate objects at run-time, simply by using the **new** object constructor keyword followed by the appropriate class name above, supplying it with the appropriate constructor arguments.

For example, If the mission includes the following XML code, a new XML object, x, is created at start-up.

```
<?RULE UK.ac.uwe.ics.followme.scripting.ScriptXML implements XML?>
..
<XML NAME="x" VALUE="y">
```

Alternatively, a script may construct an equivalent object dynamically with the following statement (but this is only available locally, not to all mission elements).

```
x = new Packages.UK.ac.uwe.ics.followme.scripting.ScriptXML('<XML NAME="x" VALUE="y">') ;
```

Missions include a number of reserved objects that are internally defined in the Mission. These include *place* (an instance of AgentPlace which represents the current location of the agent), *trader*, *pa*, *profile*, *diary*, *ispace* (the root of the pa file-system) and *myspace* (the local information space for a given agent). They are mapped to the corresponding interface of the implementing object and may therefore invoke any operation provided by the interface object directly in the mission.

Objects may be copied into the ispace from a script using an assignment. i.e. ispace.x = y ; creating a new member x of the local ispace as a copy of y. The object can be copied out by assigning the other way. i.e. z = ispace.x ; All script objects may be assigned to an information space as well as any serializable external java object.

3.6 Scripts

The script interpreter contains an implementation of the ECMAScript standard [1]. This standard was developed by the European Computer Manufacturers' Association (ECMA), and draws on a number of sources, including Netscape's JavaScript(TM) and Microsoft's JScript(TM). ECMAScript is a flexible and easy to use scripting language with syntax similar to that of Java, but is interpreted directly rather than compiled into intermediate byte-code. ECMAScript is weakly-typed in that the types of variables, functions and their parameters are not fixed. ECMAScript is also object-based rather than object-oriented, with support for classless objects. This provides an object-based programming language suitable for manipulating and controlling external systems.

The current version of the script interpreter implements a large proportion of the standard, and includes further features to facilitate agent scripting as identified in the requirements. This document describes only departures from the ECMAScript standard, this document will only describe the differences: unimplemented elements and new features. As such, the current version of this document does not intend to provide a tutorial on script programming.

The programming model

The complexity of the programming model is kept to a minimum. Whereas languages like Java provide support for multi-threading, the execution of ECMAScript is confined to a single thread. The need for scripts to respond to external events such as from a user interface or in call-backs from services, the interpreter evaluates an incoming message by *interrupting* whatever it is doing at the time. This interruption is normally invisible to the script-writer, but a script may detect the occurrence of an interrupt using the sleep() command as explained in section 3.6. In general, interrupt-handlers should be short lived so that they don't hold up the caller for longer than is necessary.

Running simple scripts

The directory, `<Framework install directory>/TestCode/Scripting` contains a number of example scripts which can be run. Each script contains comments that describe what the script is intended to do and what each line of script is accomplishing. A number of shell scripts are provided which may need to be adapted to suit your installation.

Differences from the ECMAScript standard

Certain elements of the ECMAScript standard are at present unimplemented. It is anticipated that the omissions listed here will be implemented in the future. Missing elements of the standard are:

- Dynamic function creation.
- **for.. in** statement.
- **Math** and **Date** objects.

A number of new elements were included to make things simpler for the programmer within an agent environment.
The following new elements are available over-and-above the ECMAScript standard and will be described in further detail:

- **console** - providing simple text-based I/O.
- **sleep** - providing delay and wait capabilities.
- **trace** - providing internal interpreter debugging.

The **console** object provides the facilities for simple text-based input and output. The following methods are available

- write(object) - display object.
- writeln(object) - display object and then finishes the current line.
- readln() - accept text from keyboard. Returns String.
- readln(object) - displays object as prompt and accepts text from keyboard. Returns String.

The **sleep** method provides a means for stopping interpretation of the script for a given period of time, or until an external message (using **eval** method) is received or is pending. It optionally accepts a real number argument that represents the number of seconds before timeout. The method returns a real number representing the number of seconds the interpreter was sleeping.

- sleep() - stop interpretation forever (or until interrupted). Returns number of seconds stopped.
- sleep(timeInSeconds) - stop until timeout (or until interrupted). Returns number of seconds stopped. If timeInSeconds is less than 0, then do not stop.

The following script instantiates an external Java class (mypackage.MyInterruptGenerator) to generate an interrupt (the syntax of external class utilisation will be discussed in the next section).

```
interrupted = false;
function interrupt() { interrupted = true; }

// Create external interrupt generator
generator = new Packages.mypackage.MyInterruptGenerator(this);
timeAsleep = sleep();  // wait till interrupted
// The following statement executed after external class calls interrupt()
console.writeln("Interrupted after " + timeAsleep + " seconds. ");
```

The Java external class might be:

```
package mypackage;
import UK.ac.uwe.ics.followme.scripting.ScriptObject ;

public class MyInterruptGenerator implements Runnable {
  private ScriptObject object ;

  public MyInterruptGenerator(ScriptObject o) {
    object = o ;
    (new Thread(this)).start() ;
  }

  public synchronized void run() {
     object.eval("interrupt();") ;
  }
}
```

The **trace** property enables the internal status of the interpreter to be displayed on the console. This is provided for debugging purposes and is not intended to provide information for users. It is a boolean property (initialised to false) that can be altered dynamically during script execution:

```
trace = true;
console.writeln(“This statement will be traced!”);
trace = false;
console.writeln(“This statement will not be traced!”);
```

New semantics

One major addition to the ECMAScript semantics enables access of Java classes from within the script. The **Packages** object represents the root of a tree of Java packages. This syntax follows the JavaScript LiveConnect definition [3]. The script can refer to any package or class on the Java CLASSPATH, including user-defined classes. Furthermore, any Java package, class or object can be assigned to a script variable.

```
random = new Packages.java.util.Random();
profile = new Packages.UK.uwe.ics.followme.profile.Profile();
utils = Packages.java.util;  // Package alias
vector = new utils.Vector();
```

The **java** object is a special case, and can be used anywhere where you would use **Packages.java**. So the following statements are identical and can be interchanged.

```
h = new Packages.java.util.Hashtable();
h = new java.util.Hashtable();
```

New syntax

One omission from the ECMAScript standard was the ability to have nested functions. The ability to have function declarations within functions has been implemented. The following script contains an inner function with the same name as a global function. The scoping for functions is the same as for variables, so execution results in the output “inner-inner-outer-outer-”.

```
function display() { console.write(“outer-”); }

function run() {
  function display() { console.write(“inner-”); }
  display();        // local scope
  this.display();   // local scope
  global.display(); // global scope
}
run();
display();
```

The advantage of this approach is that when the outer function is used as an object constructor, the inner functions are defined as methods of the new object. For example:

```
o = new run() ; // writes 'inner-inner-outer-'
o.display() ; // writes 'inner-'
```

Mobility

As of version 1.3, the agent framework supports mobile scripting. The *jump* command immediately suspends the activity of the script engine, and instructs the task agent (defined by the mission) to move to the indicated AgentPlace. On arrival, the script is resumed from the point of suspension. If the interpreter is asleep at the time of the jump (a jump may be initiated by any mission element) it will be woken up. A script can find out if it has moved by consulting *place*. The following script demonstrates an interrogation of the trader for active agent places, followed by a tour which visits each place in turn. This script can be found in the Missions folder as tour.xml, along with jumper.xml and pogo.xml which demonstrate similar abilities.

```
<MISSION>
  <?RULE "UK.ac.uwe.ics.followme.scripting.Script" implements SCRIPT ?>
  <SCRIPT><![CDATA[
    offers = trader.importOffers("/AgentPlace", null, null);
    for (i=0 ; i<offers.length ; i++) {
      jump(offers[i].getTagged().iface()) ;
      console.writeln("Hello from "+place.getIdString()) ;
    }
  ]]></SCRIPT>
</MISSION>
```

3.7 Generic XML Support

Support for generic XML is provided by the ScriptXML class (UK.ac.uwe.ics.followme.scripting.ScriptXML). This allows agent programmers to design their own set of XML elements and included them within a mission. The ScriptXML implementation provides basic programmatic access to the contents of this XML markup.

The outermost XML tag serves to associate a given piece of XML with the appropriate implementation rule. We don't use the tag to refer to the XML because this defines a 'class' of objects rather than any specific instance. Like all mission elements, they must be uniquely identified using a NAME attribute, which makes them accessible to other mission elements, such as scripts.

Attributes

Each XML element may include a number of attributes, a set of name/value pairs in no particular order. These attributes appear to scripts as named members of that object. In the XML fragment below, we can refer to a.X and a.Y which refer to the strings "foo" and "bar" respectively.

```
<?RULE UK.ac.uwe.ics.followme.scripting.ScriptXML implements XML?>
…
<XML NAME="a" X="foo" Y="bar"/>
```

This equivalence works both ways, so it is possible to assign to a.X and a.Y to change these values in the XML (note that the name of the instance cannot be changed once it has been created).

The tag names of XML elements is not completely lost, but is assigned to a special attribute called "tag". For example, a.tag equals "XML".

Elements

An XML element may contain any number of nested XML sub-elements. Because identical sub-elements may occur it is not possible to view these as attributes of the XML object. ScriptXML is actually a subclass of the ECMAScript Array, so we can access sub-elements by the position they appear in. XML sub-elements of ScriptXML are themselves of type ScriptXML. Extending the example above.

```
<XML NAME="a" X="foo" Y="bar"/>
  <DING/>
  dong
</XML>
```

We can refer to the sub-elements of a as , a[0] and a[1] so that the values of a[0].tag is "DING". XML Character data is mapped directly onto string values so the value of a[1] is "dong". As with attributes, assignment to XML sub-elements works just like a normal ECMAScript Array. We can overwrite sub-element 1, "dong", with the XML fragment <DONG/> using the following:

```
a[1] = new ScriptXML("<DONG/>") ;
```

To make accessing XML objects simpler, we have included a couple of accessor methods which pull out matching elements from this array. The first allows us to select sub-elements with matching tag names.

xmlObject.**select**(*tagName*) = *array of xml objects* e.g. a.select("DING")

The result of this selection is another array, because we may again have a number of sub-elements sharing the same tag name. The length of these arrays may be inspected at any time by looking at their **length** property. The second accessor method generalises this to any name-value pair.

xmlObject.**select**(*attributeName*,*attributeValue*) = *array of xml objects*

Finally, we can select plain text sub-elements (CDATA elements) with a no-argument version of select.

```
xmlObject.select() = array of strings
```

3.8 XML Documents

The AgentFramework provides a number of classes which allow the user to create XML documents, including form-based interfaces for user interaction. The XMLDocument class (UK.ac.uwe.ics.followme.scripting.io.XMLDocument) implements the Document interface as defined in the user access work-package (Note that the earlier FormXML has been deprecated). To use documents within an agent mission you need to associate a suitably named XML tag with the XMLDocument implementation. To do this you should include the appropriate processing instruction at the top of your XML mission file. e.g.

```
<?RULE UK.ac.uwe.ics.followme.scripting.io.XMLDocument implements DOCUMENT?>
```

The DOCUMENT element should at least include NAME and STYLE attributes, which define a name by which the document is referenced later in the script, and a STYLE which defines the name of an XSL definition. The markup representing the content of the document is limited only by the corresponding XSL definitions, that is, XMLDocument does not interpret this markup. To minimise the complexity of the XSL, our examples assume a format close to that of the target language HTML. Besides, HTML is as good as they come in terms of marking up layout, so, as we say in the UK, if it ain't broke - don't fix it. In actuality, you are limited only by your imagination - or more likely by your XSL programming skills. A simple 'hello world' document may be written as below.

```
<DOCUMENT NAME="hello" STYLE="myStyle">
  Hello world
</ DOCUMENT>
```

You've gotta have style

The simplicity of this document is more than made up for by the corresponding XSL. When the document is sent to a connection, the connection asks the document for the appropriate XSL, which in turn looks up the named style. Styles must currently be defined as part of the mission; they aren't separate files. Because XSL is simply a piece of XML, it can be inserted into a mission and implemented as UK.ac.uwe.ics.followme.scripting.ScriptXML, our generic XML element implementation. Your processing instructions should include something like the following.

```
<?RULE UK.ac.uwe.ics.followme.scripting.ScriptXML implements XSL?>
```

An XSL definition suitable for transforming the document above could be written as follows.

```
<XSL NAME="myStyle">
  <RULE><ROOT/>
        <HTML>
              <HEAD><TITLE>hello</TITLE></HEAD>
              <BODY><CHILDREN/></BODY>
        </HTML>
  </RULE>
</XSL>
```

Observe that the style NAME should match the document STYLE.

Assuming you have started a PA viewer which supplies the PA with a connection, you can see this document simply by delivering it from the task agent to the PA. The mission script may include something like the following.

```
<SCRIPT><![CDATA[
  pa.deliver(hello) ;
]]></SCRIPT>
```

Get Connected

The deliver method leaves it up to the PA to decide when best to display the document to the user. Early releases do not include the send() method which forces immediate rendering of the document, which is required for interactive documents including forms interfaces. We have included a temporary measure for those wanting to experiment with forms in the way of a connection object (UK.ac.uwe.ics.followme.scripting.io.ConnectionImpl), which conforms to the Connection interfaces defined by user access, and provides an easy to use AWT based HTML renderer, bypassing the PA. The following script creates a local AWT connection, and sends the document to it. Each document sent to the connection opens a new window on the local terminal.

```
<SCRIPT><![CDATA[
  pack = Packages.UK.ac.uwe.ics.followme.scripting.io ;
  connection = new pack.ConnectionImpl() ;
  connection.send(hello) ;
]]></SCRIPT>
```

HyperLinks

The Swing based AWT HTML renderer supports the use of hyper-links within a document. We can make "Hello world" a hyper-link anchor using the familiar HTML anchor tag with hypertext reference, HREF. Within a mission context, a hyperlink must refer to another named object within the scope of the mission - it is not a URL. A document may contain any number of linked documents. The document below contains a circular reference back to itself.

```
<DOCUMENT NAME="hello" STYLE="myStyle">
  <A HREF="hello">Hello world</A>
</DOCUMENT>
```

Of course documents have no built-in support for hyperlink anchors, we need to map them onto HTML in the XSL. We therefore add the following rule to the XSL.

```
<RULE><target-element type="A"/>
  <A HREF='="file:/"+attributeString("HREF")'>
       <children/>
  </A>
</RULE>
```

Do you submit?

If the rendered HTML contains forms elements including submit buttons, the information entered by the user must be communicated back to the mission. HTML submissions are conventionally referred back to a URL defined by the form ACTION. It is this object that in effect becomes the requested document. If you want the same form to re-appear after a submission, the ACTION should match the name of the document. If you have a number of linked forms, the ACTION should reference the next document to be loaded. All forms elements must be named so they can be referenced. Say our form includes a text input box called 'foo', when the user submits, the contents of this text box are added to the document as an attribute named 'foo'.

```
<DOCUMENT NAME="form" STYLE="formStyle">
  <FORM ACTION="form">
<INPUT NAME="foo" TYPE="text" SIZE="10" VALUE="anything"/>
  </FORM>
<?DOCUMENT>
```

The additional style rules needed to support this form should be added to the XSL.

```
<RULE><target-element type="FORM"/>
  <FORM ACTION='=attributeString("ACTION")'  METHOD="GET">
       <children/>
  </FORM>
<RULE>

<RULE>
<target-element type="INPUT">
  <attribute name="TYPE" value="text"/>
</target-element>
  <INPUT TYPE="text" NAME='=attributeString("NAME")' SIZE='=attributeString("SIZE")'
VALUE='=attributeString("VALUE ")'/>
</RULE>
```

Other forms elements can be used, and a complete example can be found in appendix F (XML Document example). The property values assigned to the document are as defined in HTML. Single value selection and text-area elements work in much the same way as text input boxes. Radio buttons are grouped by giving them the same name, and only the value of the button selected is returned. For checkboxes where more than one element in a named group may be selected, the connection collects the selected values together in a comma separated list. Submission buttons may also be grouped together by name, and the value of the clicked button is returned.

The mission script needs to be informed when a submission takes place, so we model script-document communication on the *Observer* pattern, so that the script is updated whenever there is a change in the state of the document. The following script adds itself as a document observer and implements the required update handler.

```
<SCRIPT><![CDATA[
  function update(observable, arg) {
       console.writeln(form.foo) ;
  }
  pack = Packages.UK.ac.uwe.ics.followme.scripting.io ;
  connection = new pack.ConnectionImpl() ;
  form.addObserver(this) ;
  connection.send(form) ;
]]></SCRIPT>
```

Communicating with the user

Many of the examples used in this document use the console as a way of communicating with the user. While this is fine for simple demonstration scripts, it isn't appropriate for fully distributed places and mobile agents. Because the job of finding the user may become quite involved, this task is handled entirely by the personal assistant which maintains a diary of the possible locations the user may be found. To communicate with the user, the agent need only talk to the PA.

We can think of user communication as being either interactive or non-interactive. For non-interactive reporting, you should use the deliver*(document)* method on the PA, which may store the document for later delivery. For this reason the delivered document should work independently of the agent mission. Other than the appropriate document style, it should refer to no other mission element. Interactive communications allow far more flexibility in terms of hyper-linked documents and forms based user input. For interactive documents use the send(*document*) method on the PA. Send() returns a boolean indicating whether or not the document was sent successfully.

We have contact

Agent initiated dialogue is only one half of the story. To allow user-initiated dialogues we make our agents *Contactable*. If you select (double-click) an *active* agent from a suitable PA viewer, it should allow you to open a new connection with the running agent. The agent must be programmed to accept user contact by implementing the contact(*connection*) function in one of its scripts. Within this function, the agent may send whatever documents are required down the connection to the user, and by adding suitable document observers, the user may effect action within the agent itself.

4 Personal Profiles and Diaries

4.1 Introduction

Many services we use require a core set of information, personal information about who we are, where we live, and how we can be contacted. All too often we find ourselves entering the same information over and over again. If only that information were stored in some common and easily accessible place. In FollowMe, this personal data is stored in the your personal profile located in your private information space. Your profile must be private because it contains sensitive personal information, only agents acting with your authority and with a reference to your information space can access this data and supply it to services as required.

Because the information in the profile object is primarily for sharing, we use XML as a platform independent means of sharing this structured data. Content markup languages like XML allow us to describe the content of an object in terms of its significant conceptual entities; which is distinct from the low-level, platform dependent object serializations used to transport agents from place to place.

The domain models for personal profiles and the personal diary were based on the vCard (used in Netscape Communicator) and vCalendar specifications, but are based on open standards using the eXtensible Markup Language, XML. The translation into XML followed the simple guideline that each element should be available in either unstructured human readable, or structured machine readable formats, or both. The unstructured form of an address could be printed directly on a label for example, whereas the structured machine readable version would be broken down into separate address fields, making elements like the post-code easy to extract and check.

4.2 Personal Profiles

The domain model for a personal profile contains the following elements:

- Identification properties (who am I?)
- Addressing properties (where do I live?)
- Communication properties (what’s my telephone number?)
- Organisational properties (what is my job?)

Personal profiles are designed to be stored as objects in the user's own private information space. Profiles may also be loaded and saved in a text based format using the eXtensible Markup Language (XML).

The <Framework install directory>/TestCode/Profile directory contains a number of test examples for you to test your installation of the information space, and the personal profile.

- ISpaceTest - You may run either the batch or shell-script file to load an example XML profile. A profile object is created and placed in the information space.

- ProfileTest - Instantiates a Profile from an (XML) text file, performs simple transformations and writes the changed profile back as a (different) text file.

- StorableProfileTest - Instantiates a Profile stored in the ISpace from an (XML) text file, and performs simple transformations.

- ProfileViewer - This application provides an awt GUI for editing a personal profile. It requires the "codebase" property to be set if the batch file is not executed at the root of the class hierarchy. This may be something like: java -Dcodebase=../../classes ProfileViewer example.xml.

4.3 Diary

Whereas the personal profile is essentially a passive information object, the personal diary is altogether different, able to raise events which rouse agents into action. The diary is composed of the following time-based elements:

- Journal - a historical record of time-stamped occurrences.
- Alarms - a single or repeating occurrence that raises a diary event.
- Events - an occurrence of a finite duration that may be associated with an alarm.
- To-Do list - an action to be performed in the future.

The personal diary provides the mechanism by which a personal assistant may relay any communications it receives, to a mobile user. If you know you will be driving on the motorway at a certain time you may want to receive traffic reports from a traffic monitoring service. Your agent is dispatched to the traffic service and you make a note of your portable messager number as a diary event along with the times you may be on the road.

Separate diaries may also be carried around by individual agents. The problems with the millenium bug pale into insignificance beside the problem of getting a mobile diary to work reliably, moving between time zones onto computers with unsynchronised clocks, without ever dropping a scheduled event. Many long-running agents do not need to be active all the time. A diary alarm can be used to wake up the agent at the specified times.

The personal profile and diary together make up a kind of electronic filofax. Used in conjunction with the personal assistant and task-agents, they enable many interesting behaviours.

Alarms

Work package E (Personal Profiles) identified the need for temporally-triggered events within the agent framework. As part of the Diary entity, an Alarm object was defined that could trigger other objects at previous defined (and possibly repeating) instants in time. This section explains how to use the early implementations of the Alarm object.

The Alarm object is a super-class of the Journal object, which represents an individual time-stamped instance cf. a diary entry such as “1pm - contacted Steve about system integration”. An alarm adds recurrence functionality (meaning the alarm may trigger multiple times following given rules) and the ability to trigger external objects. This triggering utilises an event-based system, with alarms despatching a DiaryEvent to all listeners registered to that alarm.

Alarm attributes

Each alarm contains a number of properties:
- text - a textual description of the alarm.
- start - the time associated with the alarm.

- attachment - information passed to triggered objects.
- occurrence rules - recurrence rules for when the alarm is to (re-)occur.
- exception rules - exceptions to the above occurrence rules.

The XML definition is given in Appendix D.

Format of time definition

The format for representation of dates and times is close to the ISO 8601 international standard [4]. However, several important problems have subsequently been identified with the standard and before rectification close adherence to ISO 8601 had to be avoided. One major issue with the standard is that it allows the century to be omitted from the year specifier (eg. "98" is assumed to be "1998", but what of "01"?). The format adopted is based on a technical report from the W3C [5]. Future implementations may incorporate modifications for different time zones.

YEAR (eg "1998")
YEAR-MONTH (eg. "1998-1" or "1998-01")
YEAR-MONTH-DAY (eg. "1998-1-31")
YEAR-MONTH-DAY T hour (eg. "1998-1-1T12" is midday on the 1st Jan 1998)
YEAR-MONTH-DAY T hour:min (eg. "1998-09-09T12:23")
YEAR-MONTH-DAY T hour:min:sec (eg. "1998-1-1T1:01:12")
YEAR-MONTH-DAY T hour:min:sec.s (eg. "2004-11-21T12:01:54.6")

where:

- YEAR is complete year specifier ("98" is assumed to be in the 1st century AD)
- MONTH is 1 or 2 digit (1 to 12 eg. January is "01" or "1")
- DAY is 1 or 2 digit day of month (1 to 31)
- hour is 1 or 2 digit hour of day (0 to 23)
- min is 1 or 2 digits representing minute(0 to 59)
- sec is 1 or 2 digits representing second (0 to 59)
- s is 1 or more digits representing a fraction of a second

Alarm recurrence

The Alarm has been designed to enable temporal triggering to reoccur following specified recurrence rules. This enables construction of alarms such as "trigger at 5:07pm on all weekdays in 1998, excluding 25th December and all days in June". However, the current Alarm implementation does not provide this facility and is a single-shot alarm with one trigger time associated.

5 Service Interaction

5.1 Introduction

The main focus of the agent framework is to provide an easy to use mobile agent framework that builds upon the mobile object workbench. This supports the development of simple scripted agents that can roam the web and communicate with their users through user access. However, this tells only half the story. In order for agents to be effective, they must have some way to interact with their world, and they do this via services.

The service interaction work-package addresses the problems of identifying, designing and implementing services in relation to the agent framework. Its scope ranges from the practical problems you may encounter in getting a simple service going, to more speculative issues including behavioural semantics and 'service discovery'. As such it is a cross between a 'how-to' manual and a 'what-next' guide for future research.

The distinction between agents and services forces us to think about the different roles that people may assume within the agent framework. While an Administrator is responsible for managing agent places and user accounts, it is the users themselves that launch agents and receive their results. Agents have to be written by somebody, so we may also identify the role of agent programmer. Of course, any given person may wear as many of these hats that as will fit on their head, so we may find end-user programmers responsible for their own installation. The addition of services introduces the new role of service provider, the person responsible for providing and maintaining any given service. Again, it is often convenient for the service provider to develop their own agents as clients to their services. If the service interface is generally accessible and documented, the service is open for third parties to develop their own clients and value-added services.

The agent framework provides a simple model for building programmatic agents based on the JavaScript language (codified in the ECMAScript standard, ECMA262). Central to the idea of scripting is the notion of component 'glue'. A component is generally understood to be a dynamically replaceable chunk of software that conforms to some open interface standard. These components encapsulate standard or computationally intensive sections of code to which scripting is unsuited. Scripting, combined with content markup, is suited to assembling and controlling systems of such components. In the followme system we understand these components to be intergral parts of the agent. Indeed the agent itself can be thought of as a component aggregate. When the agent moves its components move with it, so it is important that these components are subject to the same mobility requirements as any other mobile object. Services, on the other hand, do not fall under the aegis of the agent, but are separately maintained by the service provider. Whereas components are mainly created at the time an agent is launched, references to services are resolved during the lifetime of the agent by the engineering levels of the system. In followme, remote references to services are maintained transparently by the Mobile Object Workbench. In other (non-mobile) architectures, such as CORBA (Common Object Reference Broker Architecture), the initial service binding would be established by an ORB (Object Reference Broker), and subsequent communication would be supported atop IIOP (Interoperable Inter-Object Protocol).

Whereas Object Reference Brokers (ORBs) permit simple lookup of services, the trader supplied as part of the agent framework, provides comprehensive facilities for locating a service using specific search criteria. This is the enabling mechanism of service deployment supporting server-side load-balancing. Aside from their pure function, services may be classified according to other criteria of interest to the user. The pilot applications specifically anticipate the use of location as significant. It makes little sense for a regional event server to record global events, so the coverage of such a service could be recorded explicitly at the trader. In this way, an agent accessing the service could request the nearest event server to the physical location of the user at the time.

5.2 Service Interfaces

The development of a service interface involves three basic parts. The first is the identification and naming of a specific interface type. This is the primary handle by which a service can be identified. The second part is the provision of a service implementation which conforms to that type. If the type is fairly generic, a service provider may be implementing to a pre-existing type. While these two parts are sufficient for interacting with well-known interfaces and pre-built clients, the dynamic nature of the agent framework requires additional levels of description. The term computational reflection refers to the ability of software to describe itself and is the cornerstone of the scripting engine's ability to dynamically construct methods against Java components. While reflection is performed directly within the java runtime, the introspective mechanism for describing service interfaces is different because of the potential for non-java based service implementations.

Service interfaces are commonly described using a standard Interface Definition Language (IDL). CORBA IDL is the industry standard for a heterogeneous mix of clients and servers, while java services can be integrated into this architecture with Java IDL or a range of third-party products. IDL is either written, or generated from the code itself prior to execution, so is not a runtime activity like java reflection. The IDL itself can be thought of as meta-data so immediately we can see parallels between IDL and the XML language used throughout the FollowMe project; we will return to this analogy later. To make this IDL meta-data available to potential clients, it is often made available from within an interface repository, and CORBA systems provide a programmatic interface to this repository (the dynamic invocation interface, or DII). Instead of providing this kind of programmatic interface to the IDL, this document explores the common ground between IDL and XML and considers the potential of representing this meta-data within XML.

5.3 The Trader

The scope of the service interaction work-package includes identifying suitable patterns by which services can be made available to followme agents.

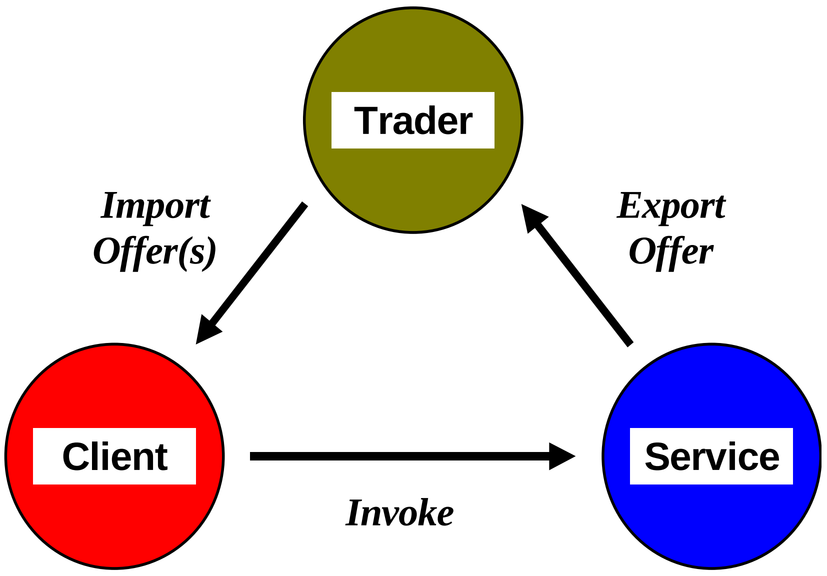
The starting point for all service interaction is to establish a link to the service. The TrivTrader supplied as part of the Mobile Object Workbench provides a basic name lookup service. This supports fundamental bootstrapping processes whereby a number of well-known services can be located. The function of the TrivTrader provides some of the basic functions of a so-called Object Request Broker, or ORB.

In the wider context of service interaction we need to provide more flexible ways of locating services. In its basic incarnation, the concept of a trader allows us to find services by their type as defined by a service interface. In addition, the Trader allows us to associate a number of properties with any given service offer. This approach allows any number of service providers to provide similar and competing services. The client is able to locate all the services providing the required service and to choose among them.

The Trader is central point of contact within the Agent Framework. It allows Agent Places and PAs to be located based on the properties of the objects. It is based on an extensible context notion which allows built-in (pre-defined) contexts and custom added contexts to coexist in the same hierarchy. The following sections introduce how Trading in FollowMe may be realised and include some code snippets.

Trading within the Agent Framework

Trading is the well established practice which allows services to advertise themselves to clients via a third-party known as a Trader. More precisely, services *export* an offer of service to the Trader. The Trader organises these offers in a structured manner allowing clients to *import* offers based on their properties leaving the clients to *invoke* any operations provided by the service. The diagram below illustrates this.



An offer defines characteristics of a service and provides a mechanism through the service may be utilised. It normally consists of the definition of the computational interface provided by the service and a set properties which are used to describe the characteristics of the service.

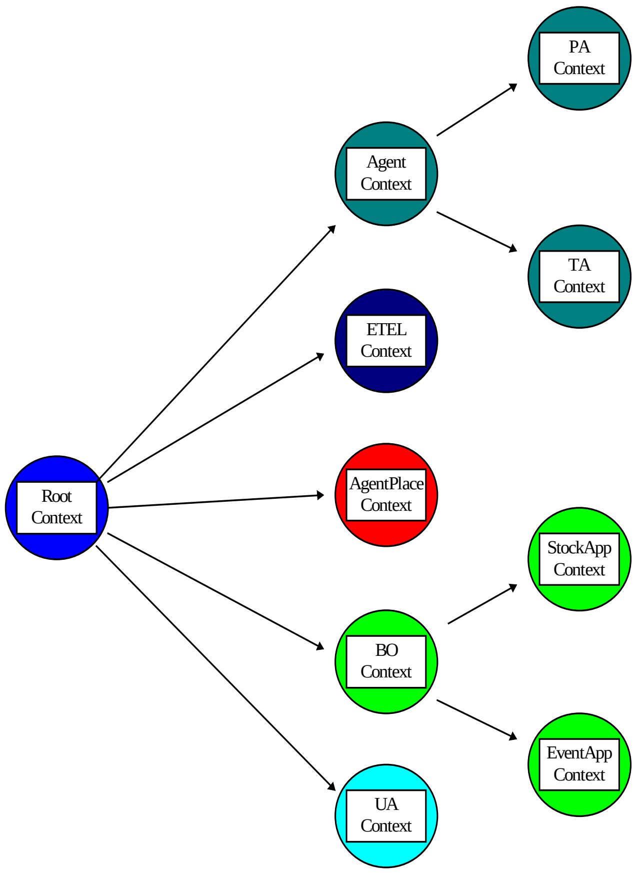
Most distributed systems require some way of finding objects in the network. Naming is one way where a readable name is bound to the object which may then be used to communicate with that object. The naming function is provided by a Name Service which implements a global name-space for the system. Trading is similar to naming in that they both allow objects to be found, however, trading allows objects to be found based on the type of service an object provides. Thus, importing a name is similar to looking a persons telephone number up in white-pages and trading equivalent to locating a person in the yellow-pages through the type of service that they offer.

In Flexinet/MOW, the TrivTrader provides a basic naming service which allows possibly remote objects to communicate. This is fine for bootstrapping an application but does not allow objects to be discriminated upon and found through the characteristics of service they provide. This provided by the Agent Framework Trader (which does however, use the TrivTrader for bootstrapping.

The Agent Framework Trader (or Trader) implements two interfaces; `Trader` and `TraderAdmin`. `Trader` implements the operations needed by clients to query the Trader for offers while `TraderAdmin` provides administration functionality is used by service exporters to customise the Trader. These interfaces are exported to the TrivTrader as "`FollowMeTrader`" and "`FollowMeTraderAdmin`" respectively. Precise details of the interfaces are provided in the "javadoc" accompanying the release software.

The `TraderAdmin` Interface

The Trader implements a hierarchical *Context Space*. The Context Space is populated with Context objects which are containers offers and missions. Additionally, the context is defined by a *Context Profile*. A Context Profile contains a description of the context, the minimum interface that is required of services in the context, and context creation details. A built in *root context* is provided by the Trader which is root of the hierarchy. The diagram below illustrates a possible context hierarchy. Note the sub-contexts of Agent and Bavaria Online (BO) which allow grouping of similar contexts.



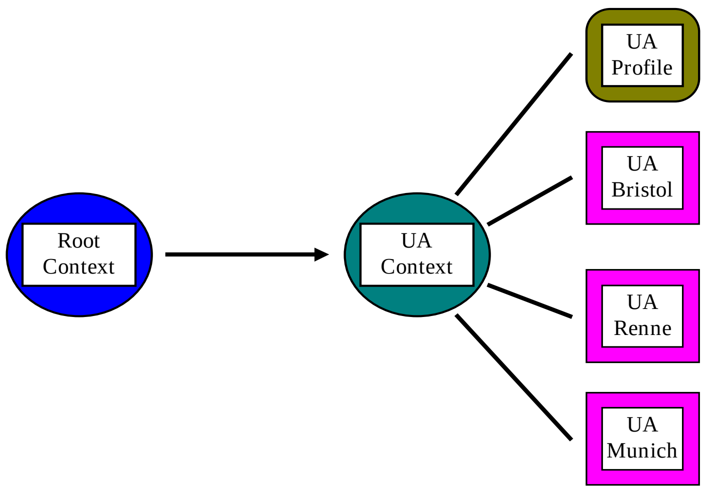
The TraderAdmin interface allows the creation and removal of contexts, adding a Context Profile to a context and adding a mission to a context. The context hierarchy is referred using a similar notation to that used by file systems. To refer to the ETEL context, the expression would be "`/ETEL`" and the StockApp context would be "`/BO/StockApp`". To reference the root context use "`/`".

Note that in order to export an offer to a context, the target context must have a context profile defined to allow interface and property checks to be made. The context profile currently defines the interface class, a text description of service offered by offers in the context, creator details and the permissible properties that may be defined by an offer and their modes. Modes may be Mandatory or Optional and Static or Dynamic; a mandatory property must be supplied by a service exporting an offer to the context, an optional property does not have to be defined. A dynamic property means that the property value may be changed by the service, a static property once defined, may not.

An example of how to use the TraderAdmin interface is in TraderAdminTest.java. The example creates a number of contexts and creates a context profile.

The Trader Interface

The Trader interface provides operations which allow clients to import offers based on an optional constraint specification, export offers, modify offers and traverse the context hierarchy to discover new services.



The diagram above shows a User Access context which contains a profile and three offers of service. In order to obtain all offers of service from the User Access the following can be used (try and catches have been omitted for clarity and it is assumed that the Trader interface has already been retrieved from the TrivTrader…see example in `<Framework install directory>/TestCode/Trader/TraderTest.java`):

```
   OfferInfo[] offers = trader.importOffers("/UA", null, null);
```

The argument "/UA" indicates that the User Access context should be searched for offers and all offers should be returned. Another form of importOffers is provided which allows a random offer to be returned if more than one offer is found to match the constraint and uses a Policy. Policies can be used to change the default behaviour of the Trader and two policies exist for importing offers: `Policy.RESOLVE_ALL` and `Policy.RESOLVE_RANDOM`. `RESOLVE_ALL` is the default. The following code illustrates the use of the `RESOLVE_RANDOM` policy to return a random UA offer.

```
   Policy[] policies = new Policy[1];
   policies[0] = new Policy(Policy.RESOLVE_RANDOM);
   OfferInfo[] offers = trader.importOffers("/UA" ,null,  policies);
```

To retrieve a specific offer, say a User Access in Bristol a property constraint string must be specified. A constraint string allows a client to specify constraints which are evaluated against the properties of the exported offer. A property consists of a name-value pair or a name-type pair. The name is a string and the currently supported types are `String`, `Boolean`, `Short`, `Long`, `Float` and `Double`. A constraint string consists of one or more logical expressions which are evaluated for candidate offers and if evaluated to true means that the offer satisfies the constraint. Constraints may contain logical `and`, `or` , ~ (not) operators and expressions may include ==, !=, <, <=, >, >= which must be appropriate for the type being evaluated otherwise an exception will be thrown. So, to get a specific User Access, the following may be used:

```
   OfferInfo[] offers = trader.importOffers("/UA", "Location ~ 'Bristol'",
   null);
```

If the context profile defines a richer set of properties including the capabilities of User Access services described in terms of boolean types, the following is possible:

```
   Policy[] policies = new Policy[1];
   policies[0] = new Policy(Policy.RESOLVE_RANDOM);
   OfferInfo[] offers = trader.importOffers("/UA",
   "Location ~ 'Bristol' and (Fax or SMS)", policies);
```

This fragment would retrieve a single offer where the location is Bristol and has Fax or SMS facilities.

Some example code which uses the Trader is in <Framework install directory>/TestCode/Trader/TraderTest.java.

Running the Trader[1]

In order to run the Trader, the MOW TrivTrader must be running; start this as per the MOW instructions. The FollowMe Trader may be started with the following:

```
java -Dflexinet.trader=(xxx)(x) UK.ac.uwe.ics.followme.trader.TraderImpl
```

where `(xxx)(x)` is the address at which the TrivTrader is listening. The classpath should be set to include the MOW class and the AgentFramework classes.

5.4 Service Profiles[2]

A simple utility class, Exporter.java, provided as part of this work-package allows a service-provider to export a service, implemented as a java class and supporting interface, to the trader. Exporter instantiates the service and ensures that it stays resident. In effect it acts as an implementation repository. Assuming the TrivTrader and TraderImpl have already been initialised, Exporter is invoked with the following command line, with the class-path set up as appropriate:

```
java UK.ac.uwe.ics.followme.scripting.si.Exporter CONTEXT INTERFACE [PROFILE [CLASS ARG* ]]
```

The CONTEXT is the name of a new or existing context that the service offer is to be registered against. The INTERFACE is the name of a class file which all services in this context must conform to (do not include the .class extension). The Service PROFILE lets us specify the properties defined within this context, and if no properties are defined in this context, the PROFILE argument may be 'null'. If we are using the Exporter to instantiate a service then the PROFILE must also include the particular values to be associated with this service. In this case, CLASS defines the service implementation. When the CLASS is instantiated, we may supply it with any number of optionally defined string arguments, ARG.

Appendix H demonstrates how to write a simple service interface and corresponding implementation, along with a client that interacts with that service. The java may be compiled and the service exported to the trader using the Exporter. The Client locates a well-known trader using the TrivTrader, and then searches for a suitable service by its type, or signature.

Once a service has been located, that reference may be used indefinitely until either the client has finished, or the service is terminated. The location transparency mechanisms provided by the Mobile Object Workbench allow both client and service to move transparently during the lifetime of the interaction.

The trader provides a programmatic interface which allows service providers to associate properties with a given service. Whenever a service is exported to the trader it must be registered within a particular trader context. The context has a corresponding profile indicating which properties can be used within that context. All services within the same context must provide the same set of properties. The Exporter simplifies the process of creating service profiles by allowing the service provider to specify the profile in XML.

We can use the Exporter to define a simple named context. The (abstract) service profile can be defined as below. Appendix I provides a Document Type Definition (DTD) for service profiles.

```
<profile desc="simple named context" creator="steve">
  <property name="name" type="STRING" mod="STATIC" req="OPTIONAL"/>
</profile>
```

The description and creator attributes are optional, and the property attributes are set here to their default values, so an alternative profile could simply be:

```
<profile><property name="name"/></profile>.
```

[1] The constraint parsing code is part of the JTrader distribution with minor modifications to allow it to work with the FollowMe Trader. This code is distributed under the terms of the GNU Public Software License, version 2, 1991.

[2] The service profiles part of Service Interaction requires the Sun XML parser which must be separately downloaded for evaluation from http://developer.java.sun.com/developer/earlyAccess/xml.

When the service provider exports a service offer, they do not need to provide all the declaration information. However, they can provide values appropriate to the new service. Appendix J shows a simple Shakespeare server which delivers plays on demand. Each service provides a different play, and as they are all of the same type they can be distinguished by the name of the play. We export two service offers with the Exporter.

```
java ..Exporter namedContext PlayServerIface hamletProfile.xml PlayServerImpl hamlet.xml
java ..Exporter namedContext PlayServerIface richardProfile.xml PlayServerImpl rich_iii.xml
```

The XML defintions of these plays are in the public domain and are included as examples with the Service Interaction work-package. Each service offer has its own profile with the appropriate name.

```
<profile><property name="name">hamlet</property></profile>
```

and

```
<profile><property name="name">richardIII</property></profile>
```

The client can retrieve the required play server from the trader by specifying a simple constraint in the importOffer(). To access only 'hamlet' servers the client would state "name=='hamlet'".

The service profile will be used later in this document to store additional meta-data relating to the service.

5.5 Service patterns 1: Stateless services

The simplest model of service interaction we can use is that of a stateless service. Each service may be associated with any number of clients, and over the course of any interaction with the service a client may pass many messages to the service. If these messages are independent of each other, the service does not have to keep track of the clients it is serving.

Stateless services may even support a simple event model. In the normal course of events a given operation should perform its work as efficiently as possible, returning control to the caller as quickly as possible. Because method invocation is synchronous, the service may hold onto the thread of control, blocking the caller. It is possible therefore to make the return of control conditional upon some critical event. The caller then knows that immediately after the call, that this event has occurred.

IDL meets XML

The specifications for the service interaction (technical annex) work-package distinguish between basic service interaction and so-called meta-level service interaction. While the former covers the mechanics of making services available within a distributed system, a meta-description of a service describes its signature. A service signature permits a form of static type checking, ensuring a good fit between client and server. More significantly within the context of the FollowMe project, it provides the raw material for the creation of new clients against a service interface.

An Interface Definition Language Language (IDL) allows the programmer to define interface signatures independently of any specific programming language. We can define the operations available at the interface including the types of parameters and return values. IDL's parameter definitions don't aim to be completely generic, but is targeted at defining simple data structures. These structures let us use sequences, structures (records), union and enumerated types. These types may also be nested, allowing arbitrarily complex data structures to be defined. There would be very little merit in translating the concepts available in IDL, directly into XML. Both may be seen as markup languages so nothing would be gained. However, we are interested in reusing the principles that IDL uses to define the interface signature.

When an agent interacts with a service, there is no reason (in principle) why the service cannot be implemented within a programming language different to the agent. As much as any company, X, would like to achieve, "Y everywhere" (substitute company and product of your choice), the real world is much messier and involves heterogenous networks of new, and legacy systems. The IDL principle, or pattern that we want to reuse here is the ability to pass information in a language neutral way. At the level of the agent framework, the JavaScript language used to build agents is weakly typed. This means that although objects know their own types, the variables they are assigned to, do not. This means that there is no way of checking at compile time if the parameters (or return types) of a scripted operation match the service signature. Whether or not you see this as a weakness or a strength of scripting, the concept of compile time type checking is redundant. Instead we should look to the actual values passed and test them to see that they conform to the requirements of the interface.

well-formed XML

A type-checking system based on XML assumes that we never pass structured types as a collection of objects as would normally be the case. We will instead assume a single generic string type that can be passed between agents and services of any persuasion. Where we adopt this pattern of service interaction we will interpret the string as containing content markup. In keeping with other FollowMe work-packages we adopt the use of XML as the specific content markup language. Our requirement is that this markup is well-formed with respect to the definition of the service interface.

The first line of defence between an agent and service returning XML data is to check that the XML string is well-formed. That is, it forms a syntactically correct XML string. A number of new classes are defined within the service interaction work package to provide different levels of checking. The first is an XML class (UK.ac.uwe.ics.followme.scripting.si.XML), which can be used to instantiate XML objects within a mission but also provides simple checking forwell-formed XML strings.

The example shown in the mission below, defines a mission which includes a simple fragment of XML. The XML is deliberately faulty; missing a closing tag, </PLAY>. The operation, isWellFormed(), carries out a simple test on the resulting XML object to see if the XML string it was constructed with is well formed.

```
<MISSION>
  <?RULE UK.ac.uwe.ics.followme.scripting.Script implements SCRIPT?>
  <SCRIPT><![CDATA[
    si =  Packages.UK.ac.uwe.ics.followme.scripting.si ;
    hamlet = new si.XML("<PLAY><SPEECH><SPEAKER>BERNARDO</SPEAKER><LINE>Who’s
there?</LINE></SPEECH>");
    if(!hamlet.isWellFormed()) console.writeln("Something is rotten in the state of XML");
  ]]></SCRIPT>
</MISSION>
```

validity

XML has inherited the use of Document Type Definitions from its SGML ancestry. The DTD is an XML document that specifies the permissible structure of some other XML document. It can be used in this context to check the validity of passed parameters and returned values.

The example shown in appendix K includes an XML file, an excerpt from Richard III, and a DTD which can be applied to it. The DTD class (UK.ac.uwe.ics.followme.scripting.si.DTD) implements a validating parser, providing a simple method, validate(), which can be applied to an XML string. In the context of a service interaction the XML string may be a value returned by a service. Before accessing the XML, or instantiating a local object based on its content, the programmer can test the string for well-formedness, and validity with respect to a given DTD. Note that the DTD itself is wrapped by a CDATA (character data) section to prevent it being interpreted directly as a DTD.Where ’richard’ represents the play, and ’play’ represents a DTD object, the statement, play. validate(richard), returns a boolean indicating if the XML is valid.

As well as a way of type-checking, the use of DTDs to represent parameter and return types is a convenient way to document the service interface. The DTDs may then be used in a constructive role to inform the agent programmer what kinds of values to expect and to provide clues as to how they should be interpreted.

In the above example the DTD used to verify the correctness of the received XML was stored in the mission itself. As the DTD relates to a specific service interface it makes more sense to store the DTD as part of the service profile. We want to use the trader as an interface repository. To do this we need to borrow some basic structure from CORBA IDL. Recall that we don’t need to translate all of the IDL specification into XML because of the significant overlap of functionality with DTD’s, all we need are a few markup elements to describe the operations available at an interface and its parameters and return types.

We will use the Exporter program again to store this meta-data as part of the service profile. The first thing we need to be able to do is to store type definitions, the equivalent of CORBA typedef’s. As the Exporter knows only about properties, we have to declare the type definition as a new property. The following service profile includes a property called ’speech.dtd’ which declares the type of the value returned from the getSpeech() operation on the PlayServer. Again, Notice that the DTD itself must be enclosed within a CDATA section to avoid confusing the XML parser.

```
<profile desc="named context with signature" creator="steve">
  <property name="name">hamlet</property>
  <property name="speech.dtd"><![CDATA[
    <!DOCTYPE SPEECH [
      <!ELEMENT SPEECH   (SPEAKER+, (LINE | STAGEDIR | SUBHEAD)+)>
      <!ELEMENT SPEAKER  (#PCDATA)>
      <!ELEMENT LINE     (#PCDATA | STAGEDIR)*>
      <!ELEMENT STAGEDIR (#PCDATA)>
      <!ELEMENT SUBHEAD  (#PCDATA)>
    ]>
  ]]></property>
</profile>
```

The SPEECH type is a structured assembly of SPEAKER, LINE, STAGEDIR and SUBHEAD elements. When the client accepts a service offer, they may retrieve the type declaration just as they retrieved the name of the service earlier. It is then a simple matter to append the contained DTD to a particular returned value and perform the appropriate type checking with DTD.validate().

This simplistic approach requires that the client knows the type name corresponding to the returned value. Clearly some additional information is needed to allow a client to locate the required types according to the operation required. CORBA IDL allows the service provider to declare the operations that are available at the interface. This operation defines the signature of a single operation. Again, unless the client already knows the names of these operations they will be unable to find the required information. Given that the client knows the name of the interface, these operations can be as below.

```
<profile desc="named context with signature" creator="steve">
  <property name="name">hamlet</property>
  <property name="speech.dtd"><![CDATA[
    <!DOCTYPE SPEECH [
      <!ELEMENT SPEECH   (SPEAKER+, (LINE | STAGEDIR | SUBHEAD)+)>
      <!ELEMENT SPEAKER  (#PCDATA)>
      <!ELEMENT LINE     (#PCDATA | STAGEDIR)*>
      <!ELEMENT STAGEDIR (#PCDATA)>
      <!ELEMENT SUBHEAD  (#PCDATA)>
    ]>
  ]]></property>
  <property name="PlayServerIface">
    <interface name="PlayServerIface">
      <operation name="getSpeech" type="speech.dtd"/>
    </interface>
  </property>
</profile>
```

The beauty of XML is that we are able to use the existing XML Document Object Model to access the content of the 'interface' property. Given the names of the interface and the required operation, we can simply extract the name of the return type, and then access the DTD as before.

The last extension we need to add to the interface signature is information about the parameters and their types. The getSpeech() operation requires three parameters, the Act, the Scene, and the Speech numbers. Because these are primitive integer types we cannot perform any type checking using DTD's which can only be used with structured objects. Although the DTD based type checking cannot distinguish between primitive types (as doesn't the script interpreter), implementers of non-scripted clients, and other service providers will find it useful to know the exact types. We adopt the basic set of primitive java types (boolean, char, byte, short, int, long, float, double), but there is no reason why this set should not be extended. Note that it is up to the reader of the service profile to interpret these keywords.

```
<profile desc="named context with signature" creator="steve">
  <property name="name">hamlet</property>
  <property name="speech.dtd"><![CDATA[
    <!DOCTYPE SPEECH [
      <!ELEMENT SPEECH   (SPEAKER+, (LINE | STAGEDIR | SUBHEAD)+)>
      <!ELEMENT SPEAKER  (#PCDATA)>
      <!ELEMENT LINE     (#PCDATA | STAGEDIR)*>
      <!ELEMENT STAGEDIR (#PCDATA)>
      <!ELEMENT SUBHEAD  (#PCDATA)>
    ]>
  ]]></property>
  <property name="PlayServerIface">
    <interface name="PlayServerIface">
      <operation name="getSpeech" type="speech.dtd">
        <parameter name="act" type="int"/>
        <parameter name="scene" type="int"/>
        <parameter name="speech" type="int"/>
      </operation>
    </interface>
  </property>
</profile>
```

5.6 Service patterns 2: Call-Back

In many cases there is a need for the service to communicate back to the client; a call-back. To achieve this, the client must also present the appropriate interface to the service. A service context must therefore be able to handle more than one service defintion - the primary service offered by the provider, and the secondary interface to the client.

We can add a new operation to the PlayServer interface which 'pushes' lines to the client, rather than the client 'pulling' individual lines as before. Call-back gives control to the service, allowing the service to control the operations of any number of clients. This scenario allows different clients to represent different characters in the play, with the service acting as director, cueing them with their lines at the correct times.

The following scenario includes two clients, BERNARDO and FRANCISCO being fed lines by the service. By using call-backs we are able to correctly sequence the lines recited by these agents. The dialogue ends when Horatio fails to appear on cue.

```
BERNARDO: Who’s there?
FRANCISCO: Nay, answer me: stand, and unfold yourself.
BERNARDO: Long live the king!
FRANCISCO: Bernardo?
BERNARDO: He.
FRANCISCO: You come most carefully upon your hour.
BERNARDO: ’Tis now struck twelve; get thee to bed, Francisco.
FRANCISCO: For this relief much thanks: ’tis bitter cold,
FRANCISCO: And I am sick at heart.
BERNARDO: Have you had quiet guard?
FRANCISCO: Not a mouse stirring.
BERNARDO: Well, good night.
BERNARDO: If you do meet Horatio and Marcellus,
BERNARDO: The rivals of my watch, bid them make haste.
FRANCISCO: I think I hear them. Stand, ho! Who’s there?
```

The service profile should document both the interface offered by the service, and the Actor interface required of each client. Because this profile defines two interfaces it may not be clear which of the two is actually implemented by the service itself. To clarify matters, an 'interface' property is added which indicates which of the declared interfaces is supported by the service offer. The additional 'Actor' interface is used like a type in the 'enter' operation on this interface.

```
<profile desc="named context with signature" creator="steve">
  <property name="name">hamlet</property>

  <property name="speech.dtd"><![CDATA[
    <!DOCTYPE SPEECH [
      <!ELEMENT SPEECH   (SPEAKER+, (LINE | STAGEDIR | SUBHEAD)+)>
      <!ELEMENT SPEAKER  (#PCDATA)>
      <!ELEMENT LINE     (#PCDATA | STAGEDIR)*>
      <!ELEMENT STAGEDIR (#PCDATA)>
      <!ELEMENT SUBHEAD  (#PCDATA)>
    ]>
  ]]></property>

  <property name="Actor">
    <interface name="Actor">
      <operation name="cue">
  <parameter name="line" type="String"/>
      </operation>
    </interface>
  </property>

  <property name="PlayServerIface">
    <interface name="PlayServerIface">
      <operation name="getSpeech" type="speech.dtd">
        <parameter name="act" type="int"/>
        <parameter name="scene" type="int"/>
        <parameter name="speech" type="int"/>
      </operation>
      <operation name="enter">
        <parameter name="persona" type="String"/>
        <parameter name="actor" type="Actor"/>
      </operation>
    </interface>
  </property>
```

```
    <property name="interface">PlayServerIface</property>
  </profile>
```

The problem with this service interface is that it is like playing with a multi-user dungeon, access to the service is global and anybody can join the cast at any time. The next step is to localise the service-state so that one or more clients can conduct a session with the service in private. We create a service handle that encapsulates the service-state. For this example, the service handle implements the same interface as the primary service, from which we can request a new handle with getHandle(). All subsequent interaction between client and service is carried out via this handle. If another group of clients wishes to interact with the service they only have access to the primary service, and should spawn off a new handle with its own internal state. Service handles allow one or more clients to interact with a stateful service as though they were the only clients. The only extension to the service profile required by this example is the new operation getHandle() which returns a PlayServerIface.

6 References

[1] ECMA, ECMAScript: A general-purpose, cross-platform programming language. ECMA Standard ECMA-262 June 1997 available from http://www.ecma.ch/

[2] FollowMe, Autonomous Agents Design, FollowMe Deliverable DD3

[3] David Flanagan, *JavaScript: The definitive guide* Second Edition. O'Reilly & Associates Jan 1997 ISBN 1-56592-234-4

[4] ISO *International Standard ISO 8601 : 1988(E) Date elements and interchange formats* 15 June 1988

[5] Misha Wolf, Charles Wicksteed, *Date and Time* W3C Technical Report TR "NOTE-datetime-970915" 15 Sept 1997

7 Appendices

7.1 Appendix A - BNF for ECMAScript

```
Literal::= NullLiteral
             | BooleanLiteral
             | NumericLiteral
             | StringLiteral
NullLiteral::= "null"
BooleanLiteral::= ( "true" | "false" )
NumericLiteral::= <NUMERIC>
StringLiteral::= <STRING>
Identifier::= <IDENTIFIER>
PrimaryExp::= "this"
             | Identifier
             | Literal
             | "(" Expression ")"
Arguments::= "(" ( AssignmentExp ( "," AssignmentExp )* )? ")"
MemberExp::= PrimaryExp ( Arguments | "[" Expression "]" | "." Identifier )*
NewExp::= "new" MemberExp
             | MemberExp
PostfixExp::= NewExp ( ( "++" | "--" ) )?
UnaryExp::= PostfixExp | ( "delete" | "void" | "typeof" | "+" | "-" | "~" |"!" | "++" | "--" ) UnaryExp
MultiplicativeExp::= UnaryExp ( ( "*" | "/" | "%" ) UnaryExp )*
AdditiveExp::= MultiplicativeExp ( ( "+" | "-" ) MultiplicativeExp)*
ShiftExp::= AdditiveExp ( ( "<<" | ">>" | ">>>" ) AdditiveExp )*
RelationalExp::= ShiftExp ( ( "<" | ">" | "<=" | ">=" ) ShiftExp )*
EqualityExp::= RelationalExp ( ( "==" | "!=" ) ShiftExp )*
BitwiseAndExp::= EqualityExp ( "&" EqualityExp )*
BitwiseXorExp::= BitwiseAndExp ( "^" BitwiseAndExp )*
BitwiseOrExp::= BitwiseXorExp ( "|" BitwiseXorExp )*
LogicalAndExp::= BitwiseOrExp ( "&&" BitwiseOrExp )*
LogicalOrExp::= LogicalAndExp ( "||" LogicalAndExp )*
ConditionalExp::= LogicalOrExp ( "?" AssignmentExp ":" AssignmentExp )?
AssignmentExp::= ConditionalExp ( ( "=" | "*=" | "/=" | "%=" | "+=" |
                 "-=" | "<<=" | ">>=" | ">>>=" | "&=" | "^=" | "|=" ) ConditionalExp )*
Expression::= AssignmentExp ( "," AssignmentExp )*
Statement::= Block
             | VariableStmt
             | EmptyStmt
             | ExpressionStmt
```

```
            | IfStmt
            | WhileStmt
            | ForStmt
            | ContinueStmt
            | BreakStmt
            | ReturnStmt
            | WithStmt
Block::= "{" ( Statement | FunctionDec )* "}"
VariableStmt::= "var" VariableDecList ";"
VariableDecList::= VariableDec ( "," VariableDec )*
VariableDec::= Identifier [ Initializer ]
Initializer::= "=" AssignmentExp
EmptyStmt::= ";"
ExpressionStmt::= Expression ";"
IfStmt::= "if" "(" Expression ")" Statement [ "else" Statement ]
WhileStmt::= "while" "(" Expression ")" Statement
ForStmt::= "for" "(" [ (Expression | "var" VariableDecList) ] (  [ Expression ] ";" | "in" ) [ Expression ] ")"
            Statement
ContinueStmt::= "continue" ";"
BreakStmt::= "break" ";"
ReturnStmt::= "return" [ Expression ] ";"
WithStmt::= "with" "(" Expression ")" Statement
FunctionDec::= "function" Identifier "(" [ FormalParamList ] ")" Block
FormalParamList::= Identifier ( "," Identifier )*
Script::= ( Statement | FunctionDec )*
```

7.2 Appendix B - Scripted Example

This appendix contains an example script to demonstrate the use of external Java classes and integration with FollowMe objects (in this case the Personal Profile of Work package E). This is a simplification of a scenario where a scripted agent might contact a user via a textual SMS message onto a cell phone. In this simple example, a Personal Profile is created from an XML text file (instead of obtained from a Personal Assistant). The cell phone details are extracted by searching the telecommunication details stored within the profile. This enables a message to be created using the users first name (also obtained from the profile) which is then sent to the cell phone via a gateway. The sending process (instead of using the Personal Assistant and User Access) is achieved using an SMS Gateway implemented as a Java class (UK.ac.uwe.ics.followme.SMS.VodafoneGateway).

```
/*
 * SMS Message Send Example
 *
 * Creates a FollowMe PersonalProfile from an XML file
 * and sends an SMS message to cell phone details contained
 * within profile.
 *
 * my & jpt 23/7/98
 *
 * (c) UWE, Bristol 1998
 *
 */
followme = Packages.UK.ac.uwe.ics.followme; // Package alias

// Send an SMS message to a given phone through a gateway
function sendMessage(SMSgateway, phoneNumber, SMSmessage){
  console.writeln("Sending ["+SMSmessage+"] to ["+phoneNumber+"]
                         using ["+SMSgateway+"]");
  args = new Array(SMSgateway, phoneNumber, SMSmessage);
  SMS_imp = followme.SMS.VodafoneGateway.main(args);
}

// Create a personal profile (myProfile) from an XML script
myProfile = new followme.profile.Profile(); // instantiate empty profile
XML_FILENAME = "myprofile.xml";
myProfile.load(XML_FILENAME);  // DEBUG method to load from file
```

```
// Extract profile user’s first name for SMS message from PP
firstName = myProfile.get_id().get_first();      // first name

// Search the profile for the first cell phone
numPhones = myProfile.length_tel(); // number of comms details
for(id=0, found=false;(id<numPhones) && (!found));id++){
  details = myProfile.get_tel(id);
  if (details.get_cell()) {  // is it a cell phone?
    number = details.get_text();      // phone number
    gateway = details.get_gateway();  // phone SMS gateway
    found = true;
  }
}
if (!found) {
  console.writeln("No cell phone details found in profile.");
} else {
  // Confirm with user that message should be sent
  reply = console.readln("Send message to "+firstName+" on "+gateway+" ?");
  if ((reply == "Y") || (reply == "y")) {
    message = "Hello " + firstName + " , this message generated
        automatically by a script using PersonalProfile data in XML -
        FollowMe is on the air!";
    // Perform the send operation
    sendMessage(gateway, number, message);
  }
}
```

7.3 Appendix C - Mission Example

This example contains a script designed to interact with an SMS gateway, embedded within a mission. The mission also contains a simple profile-like object which defines the user details. This generic XML is implemented by the ScriptXML class (see the second implementation rule). This class presents the XML to the script as an Array object, where properties defined in the opening tag appear as named attributes, and sub-elements appear as array elements. ScriptXML also provides a ‘select(tag)’ method which selects sub-elements by tag name, or ‘select(attribute,value)’ which selects elements where attribute=value.

```
<MISSION>
  <?RULE UK.ac.uwe.ics.followme.scripting.Script implements SCRIPT?>
  <?RULE UK.ac.uwe.ics.followme.scripting.ScriptXML implements
PERSONALDATA?>
  <PERSONALDATA name=’mikey’>
  Michael Yearworth’s Personal Data
  Note that this is NOT a valid FollowMe Personal Profile
  <NAME FamilyName="Yearworth" FirstName="Michael" Prefix="Dr."> Dr. Michael
Yearworth</NAME>
  <TEL Work="441179763857" Mobile="44467872873"
MobileGateway="0385499999"></TEL>
  <EMAIL Work="my@ics.uwe.ac.uk"></EMAIL>
</PERSONALDATA>
<SCRIPT><![CDATA[

function PrintSpace(lines){
 for(i=0;i<lines;i++){
  console.writeln("");
 }
}

trace = false; // turn off tracing (unnecessary since this is default!)
PrintSpace(2);
console.writeln("FollowMe Agent Framework (c) University of the West of
England, Bristol 1998");
PrintSpace(2);


//Extract user’s first name for SMS message from PersonalData
Name = mikey.select("NAME");  // Extracts Name Field
FirstName = Name[0].FirstName;  // Extracts FirstName

//Get the Mobile Phone number
Tel = mikey.select("TEL");
TelephoneNumber = Tel[0].Mobile;

//get Mobile Phone gateway number
Gateway = Tel[0].MobileGateway;
AccessCode = "9" + Gateway;

// construct personal message
SMSMessage = "Hello " + FirstName + " , this message generated automatically
by a <PPtest.fmm> Agent using Personal Data held im mission script";
Args = new Array(AccessCode, TelephoneNumber, SMSMessage);
SMS_imp = Packages.UK.ac.uwe.ics.followMe.SMS.VodafoneGateway.main(Args);


  ]]></SCRIPT>
</MISSION>
```

7.4 Appendix D - Alarm DTD

This appendix contains the document type definition (DTD) for XML versions of the diary.

```
<!-- FollowMe Diary DTD -->
<!DOCTYPE DIARY [

<!ELEMENT DIARY     (JOURNAL | ALARM | EVENT | TODO)*>

  <!ATTLIST DIARY   REVISION          CDATA #IMPLIED>
  <!ATTLIST DIARY   VERSION           CDATA #IMPLIED>
  <!ATTLIST DIARY   LANGUAGE          CDATA #IMPLIED>
  <!ATTLIST DIARY   CLASS       (PUBLIC|PRIVATE) "PUBLIC"
  <!ATTLIST DIARY   PRODID            CDATA #IMPLIED>


<!-- JOURNAL - single, instantaneous, timestamped occurrence -->
<!ELEMENT JOURNAL (#PCDATA)>

  <!ATTLIST JOURNAL START       CDATA #REQUIRED>


<!-- ALARM - possibly repeating instantaneous occurrence, that raises a DiaryEvent -->
<!ELEMENT ALARM (#PCDATA | ORULE | EXRULE)* >

  <!ATTLIST ALARM   START       CDATA #REQUIRED>
  <!ATTLIST ALARM   ATTACH            CDATA #IMPLIED>


<!-- EVENT - occurence with finite duration (eg. start & finish or duration) -->
<!ELEMENT EVENT     (#PCDATA | ORULE | EXRULE | ALARM)* >

  <!ATTLIST EVENT   START       CDATA #REQUIRED>
  <!ATTLIST EVENT   END               CDATA #IMPLIED>
  <!ATTLIST EVENT   DURATION          CDATA #IMPLIED>


<!-- TODO - action to be performed -->
<!ELEMENT TODO(#PCDATA | ORULE | EXRULE | ALARM)* >

  <!ATTLIST TODO          START       CDATA #REQUIRED>
  <!ATTLIST TODO          COMPLETED         CDATA #IMPLIED>
  <!ATTLIST TODO          DURATION          CDATA #IMPLIED>


<!-- ORULE - Ocurrence rule -->
<!ELEMENT ORULE EMPTY >

  <!ATTLIST ORULE   VALUE       CDATA #REQUIRED>


<!-- EXRULE - Exception rule -->
<!ELEMENT EXRULE EMPTY >

  <!ATTLIST EXRULE  VALUE       CDATA #REQUIRED>

]>
```

7.5 Appendix E - Alarm Example

This example contains a mission script designed to wait for an alarm triggered event. The mission instantiates the alarm from the XML source and registers the script as a listener. The script then sleeps till interrupted by the alarm. The diary event caused by the alarm is captured by the script, causing the *serviceAlarm* function to be called (this was specified in the alarm attachment).

```
<MISSION>

  <!-- A simple single-occurrence alarm -- >
  <?RULE UK.ac.uwe.ics.followme.diary.AlarmImpl implements ALARM?>
  <ALARM name = 'myAlarm'
          attach = 'serviceAlarm();'
          start = '1998-9-24T13:20'>
  A simple alarm that triggers at 1:20pm on 24th September 1998.
  </ALARM>

<?RULE UK.ac.uwe.ics.followme.scripting.Script implements SCRIPT?>
<SCRIPT><![CDATA[

// Interrupt function executed on receipt of diary event from alarm
function serviceAlarm(){
  console.writeln("Alarm has occurred!");
  // Perform alarm-specific processing
}

// Add this script as a listener for the alarm
myAlarm.addDiaryListener(this);

diaryPackage = Packages.UK.ac.uwe.ics.followme.diary;
currentTime = new diaryPackage.DateTime();
console.writeln("Current time is " + currentTime);
console.writeln("Alarm set for" + myAlarm.get_start());

console.writeln("Waiting for the alarm to occur…);
sleep();  // sleep till interrupted by alarm
console.writeln("Finished");

  ]]></SCRIPT>
</MISSION>
```

7.6 Appendix F - XML Document Example

```
<MISSION>

<?RULE UK.ac.uwe.ics.followme.scripting.Script implements SCRIPT?>
<?RULE UK.ac.uwe.ics.followme.scripting.io.XMLDocument implements DOCUMENT?>
<?RULE UK.ac.uwe.ics.followme.scripting.ScriptXML implements XSL?>

<DOCUMENT name="foo" style="myStyle" width="600" height="500">
  <A HREF="foo">this</A>
  <FORM ACTION="foo">
    hello world
    <INPUT type="text" size="10" value="foobar" name="a"/>
    <INPUT type="password" name="b" value="guest"/><BR/>
    <INPUT type="checkbox" checked="true" name="c" value="foo"/>foo
    <INPUT type="checkbox" name="c" value="bar"/>bar
    <INPUT type="radio" name="d" value="foo"/>
    <INPUT type="radio" name="d" checked="true" value="bar"/><BR/>
    <INPUT type="hidden" value="foo" name="e"/><BR/>
    <SELECT size="3" name="f">
      <OPTION>one</OPTION>
      <OPTION>another one</OPTION>
      <OPTION>foo</OPTION>
      <OPTION>bar</OPTION>
    </SELECT><BR/>
    <TEXTAREA cols="10" rows="5" name="g"/>
    <INPUT type="submit" name="h" value="one"/>
    <INPUT type="submit" value="two" name="h"/>
    <INPUT type="reset" name="i" value="reset"/>
  </FORM>
</DOCUMENT>
```

```
<XSL name="myStyle">

<rule>
  <root/>
  <HTML><HEAD><TITLE>Form</TITLE></HEAD>
    <BODY><children/></BODY>
  </HTML>
</rule>

<rule>
  <target-element type="A"/>
  <A HREF='="file:/"+attributeString("HREF")'>
    <children/>
  </A>
</rule>

<rule>
  <target-element type="FORM"/>
  <FORM ACTION='=attributeString("ACTION")' METHOD="GET">
    <children/>
  </FORM>
</rule>

<rule>
  <target-element type="INPUT">
    <attribute name="type" value="text"/>
  </target-element>
  <INPUT      TYPE="text"      NAME='=attributeString("name")'      VALUE='=attributeString("value")'
SIZE='=attributeString("size")'/>
</rule>

<rule>
  <target-element type="INPUT">
    <attribute name="type" value="password"/>
  </target-element>
  <INPUT    TYPE="password"    NAME='=attributeString("name")'    VALUE='=attributeString("value")'
SIZE='=attributeString("size")'/>
</rule>

<rule><target-element type="INPUT">
    <attribute name="type" value="checkbox"/>
    <attribute name="checked" value="true"/>
  </target-element>
  <INPUT    TYPE="checkbox"    NAME='=attributeString("name")'    VALUE='=attributeString("value")'
CHECKED="true"/>
</rule>

<rule>
  <target-element type="INPUT">
    <attribute name="type" value="checkbox"/>
  </target-element>
  <INPUT TYPE="checkbox" NAME='=attributeString("name")' VALUE='=attributeString("value")'/>
</rule>

<rule>
  <target-element type="INPUT">
    <attribute name="type" value="radio"/>
    <attribute name="checked" value="true"/>
  </target-element>
  <INPUT     TYPE="radio"     NAME='=attributeString("name")'     VALUE='=attributeString("value")'
CHECKED="true"/>
</rule>

<rule>
  <target-element type="INPUT">
    <attribute name="type" value="radio"/>
  </target-element>
  <INPUT TYPE="radio" NAME='=attributeString("name")' VALUE='=attributeString("value")'/>
</rule>

<rule>
  <target-element type="INPUT">
    <attribute name="type" value="hidden"/>
  </target-element>
  <INPUT TYPE="hidden" NAME='=attributeString("name")' VALUE='=attributeString("value")'/>
</rule>

<rule>
  <target-element type="INPUT">
    <attribute name="type" value="submit"/>
  </target-element>
  <INPUT TYPE="submit" NAME='=attributeString("name")' VALUE='=attributeString("value")'/>
```

```
</rule>

<rule>
  <target-element type="INPUT">
    <attribute name="type" value="reset"/>
  </target-element>
  <INPUT TYPE="reset" NAME='=attributeString("name")' VALUE='=attributeString("value")'/>
</rule>

<rule>
  <target-element type="TEXTAREA"/>
  <TEXTAREA             WRAP='=attributeString("wrap")'             ROWS='=attributeString("rows")'
COLS='=attributeString("cols")' NAME='=attributeString("name")'>
    <children/>
  </TEXTAREA>
</rule>

<rule>
  <target-element type="SELECT">
    <attribute name="MULTIPLE" value="true"/>
  </target-element>
  <SELECT SIZE='=attributeString("size")' NAME='=attributeString("name")' MULTIPLE="true">
    <children/>
  </SELECT>
</rule>

<rule>
  <target-element type="SELECT"/>
  <SELECT SIZE='=attributeString("size")' NAME='=attributeString("name")'>
    <children/>
  </SELECT>
</rule>

<rule>
  <target-element type="OPTION"/>
  <OPTION>
    <children/>
  </OPTION>
</rule>

<rule>
  <target-element type="BR"/>
  <BR/>
</rule>
</DOCUMENT>

<SCRIPT><![CDATA[

pack = Packages.UK.ac.uwe.ics.followme.scripting.io ;

function update(observable, arg) {
  console.writeln(foo) ;
  console.writeln("a = "+foo.a) ;
  console.writeln("b = "+foo.b) ;
  console.writeln("c = "+foo.c) ;
  console.writeln("d = "+foo.d) ;
  console.writeln("e = "+foo.e) ;
  console.writeln("f = "+foo.f) ;
  console.writeln("g = "+foo.g) ;
  console.writeln("h = "+foo.h) ;
  console.writeln("i = "+foo.i) ;
  if (arg==null) java.lang.System.exit(0) ;
}

console.writeln("opening connection") ;
connection = new pack.ConnectionImpl() ;

console.writeln("sending form") ;
foo.addObserver(this) ;
connection.send(foo) ;

while (true) sleep() ;

]]></SCRIPT>
</MISSION>
```

7.7 Appendix G - Interaction diagrams

The following interaction diagrams illustrate some typical scenarios commonly required by mobile agent based systems and illustrates how they are supported by the components shipped with the agent framework.

Create a PA

An administrator downloads a FollowMe distribution package and installs it on their local machine. Following installation, a new AgentPlace is created. In order for a user to have an agent representative, the administrator creates a new PA for them. On initialisation, the PA creates a new Information Space (IS) containing an empty Personal Profile (and Personal Diary). Finally, the PA is named and registered with the Trader.

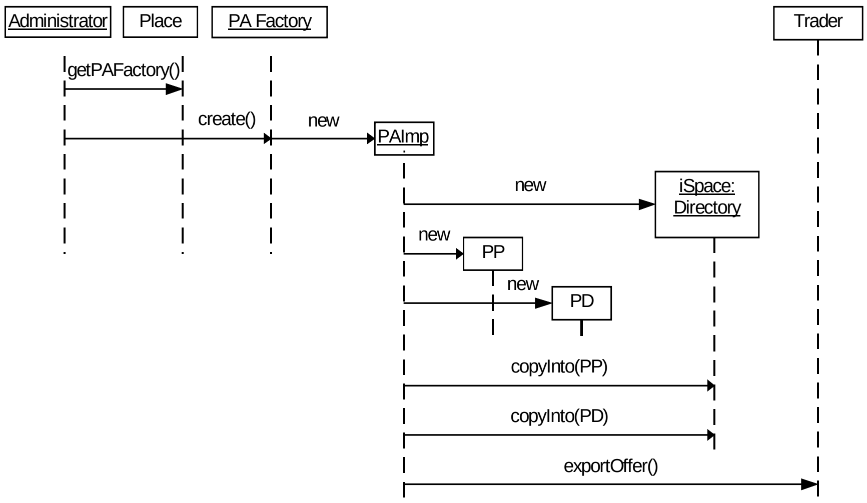


Figure 1 Create personal assistant

Contact a PA

The user opens a new connection with the name of their PA. The first screen sent back validates the user password. If successful, the user is sent the main PA view from which they may select one of the many functions available within the PA. If unsuccessful, the user is prompted to re-enter their password.

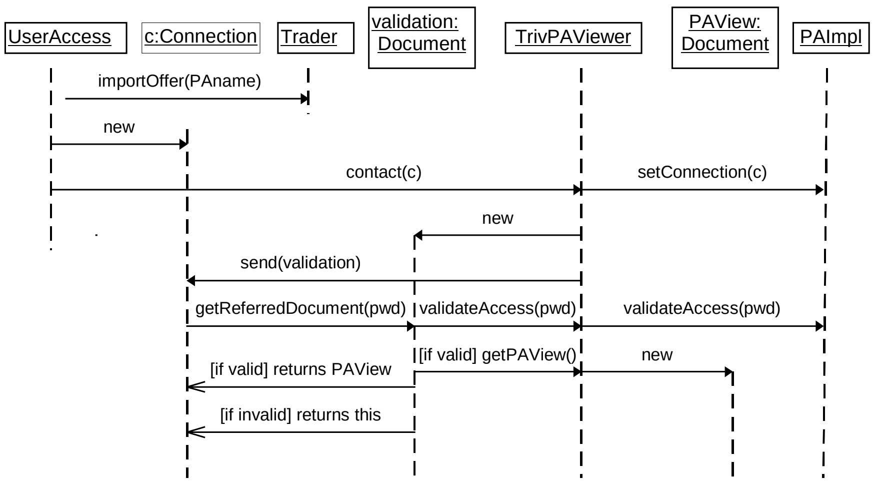


Figure 2 Contact a PA

Browse Trader

This is a function available from within the PA. The PA contacts the Trader requesting all of the missions available within the trader.

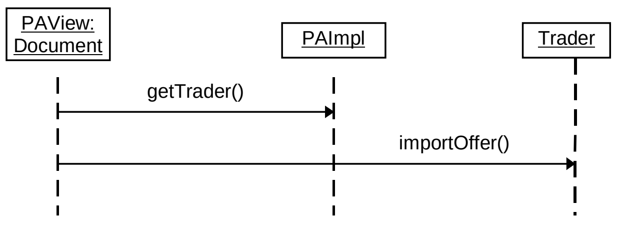
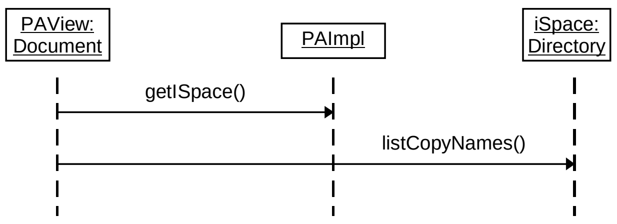


Figure 3 List services

Browse iSpace



Browse iSpace

Browse Active Tasks

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.

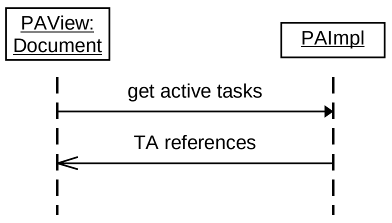


Figure 4 Select a task

Create a Task Agent

The user has selected a task agent from an available list, and they want to launch it. The PA invokes an operation on the TAFactory supplying as a parameter Mission which the TAFactory uses as a “template” for the new agent. The agent is instantiated in the current AgentPlace. The PA records the Task Agent as dispatched.

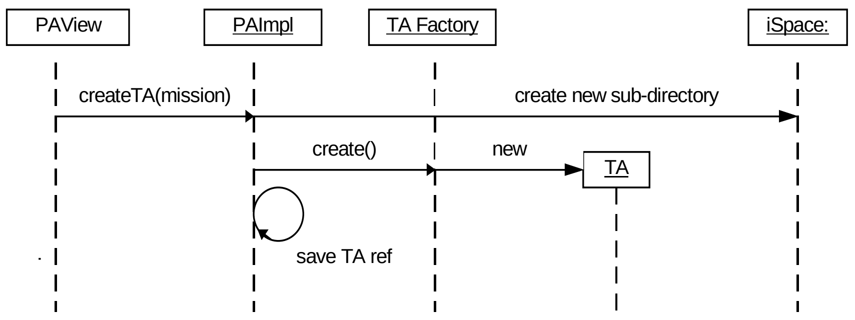


Figure 5 Create task agent

Contact Task Agent

Assuming that the user has already established a connection, the user wishes to contact the task agent via the PA.

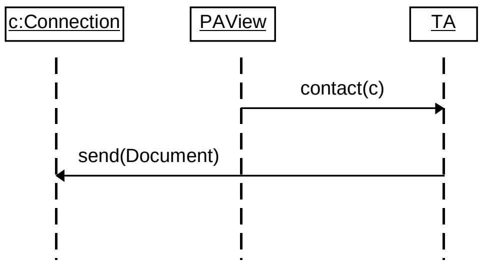


Figure 6 Contact agent

Report to User

The agent has either finished its task or needs to relay an interim report to the user. This is essentially a one-way communication initiated by the agent.

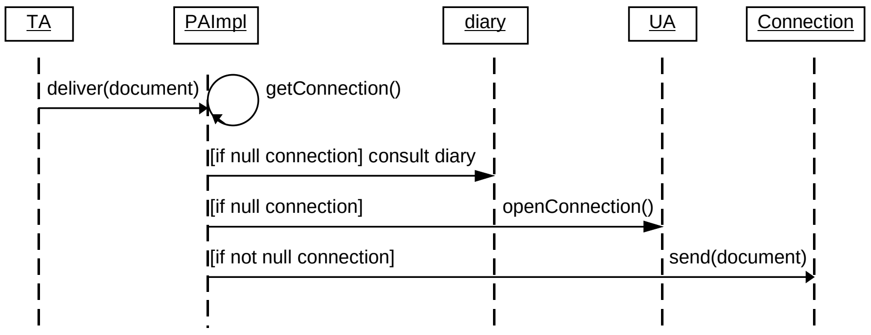


Figure 7 deliver document

Dialogue with User

Two-way communication initiated by a task agent.

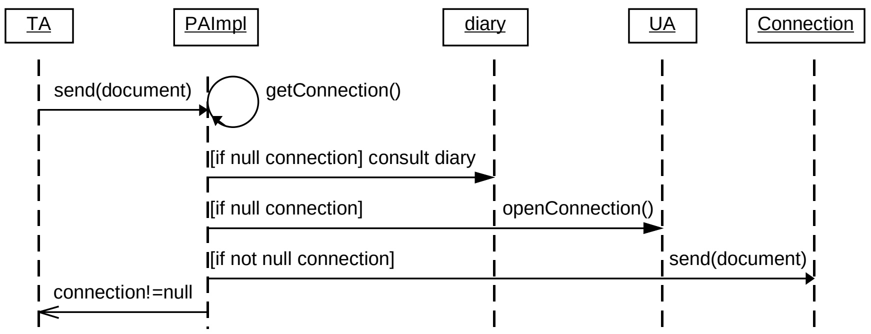


Figure 8 send document

Move PA 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.

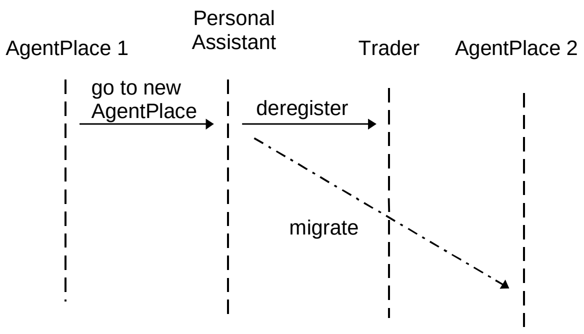


Figure 9 PA migration to remote AgentPlace

Move PA 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.

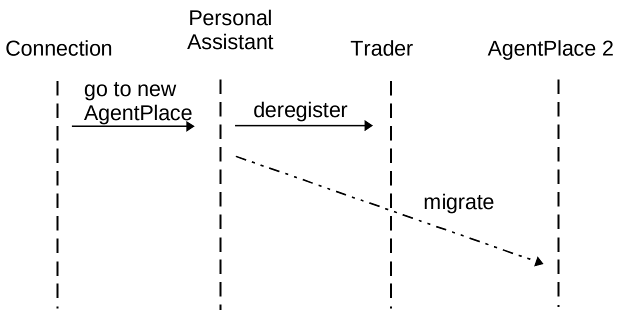


Figure 10 Recall PA from remote AgentPlace

Kill 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 invokes a kill() operation on the agent. This will prompt the PA to check whether there are any outstanding tasks; if there are it will attempt to cancel the tasks. Once this is complete, the entry for the PA in the Trader is removed, the Information Space and anything in it is removed and the PA is destroyed.

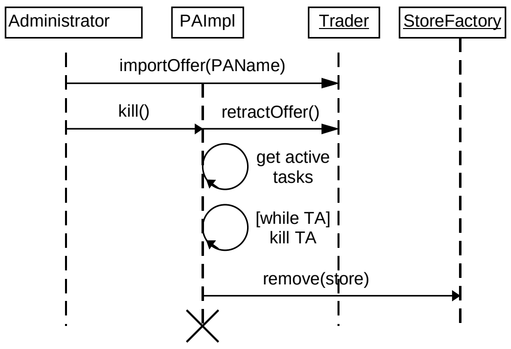


Figure 11 Kill a PA

7.8 Appendix H - "hello world" service and client.

```
/** hello world service interface */

public interface HelloWorldIface {
    public String hello() ;
}


/** hello world service implementation */

public class HelloWorldImpl implements HelloWorldIface{
    public String hello() {
  return "hello world";
    }
}

/** hello world client */

import UK.co.ansa.flexinet.core.naming.Tagged;
import UK.co.ansa.flexinet.trivtrader.FNetTrader;
import UK.ac.uwe.ics.followme.trader.TraderAdmin;
import UK.ac.uwe.ics.followme.trader.Trader;
import UK.ac.uwe.ics.followme.trader.TraderException;
import UK.ac.uwe.ics.followme.trader.OfferInfo;
import UK.co.ansa.flexinet.test.FNetTest;

public class HelloWorldClient {

    public static void main(String[] args) {
  try {
      String context = args[0];

      FNetTrader fnt = FNetTest.getTrader();
      if (fnt == null) throw new Exception("couldn’t find the TrivTrader");

      TraderAdmin admin = (TraderAdmin) fnt.get("FollowMeTraderAdmin");
      if (admin == null) throw new Exception("FollowMeTraderAdmin not resolved with
TrivTrader");

      Trader trader = (Trader) fnt.get("FollowMeTrader");
      if (admin == null) throw new Exception("FollowMeTrader not resolved with TrivTrader");

      // bind to the service
      OfferInfo[] offers ;
      offers = trader.importOffers("/"+context, null, null);
      for (int i=0 ; i<offers.length;i++) {
        Object service = offers[i].getTagged().iface() ;
        if (service instanceof HelloWorldIface) {
            // service interaction
            System.out.println(((HelloWorldIface) service).hello());
        }
      }
  } catch (Exception error) {
      error.printStackTrace(System.err);
```

```
     } catch (TraderException error) {
        error.printStackTrace(System.err);
     }
      }

   }
```

7.9 Appendix I - DTD for service profiles

```
<!-- DTD for service profile declarations    Steve Battle Feb99 -->

<!ELEMENT profile (property)*>
<!ATTLIST profile
    desc CDATA #IMPLIED
    creator CDATA #IMPLIED>

<!ELEMENT property ANY>
<!ATTLIST property
    name CDATA #REQUIRED
    type (BOOLEAN|SHORT|USHORT|LONG|ULONG|FLOAT|DOUBLE|CHAR|STRING) "STRING"
    mod (STATIC|DYNAMIC) "STATIC"
    req (OPTIONAL|MANDATORY) "OPTIONAL">
```

7.10 Appendix J – Client of stateless service

```
/*
 * PlayServerIface.java
 * Defines a simple stateless service
 *
 */

public interface PlayServerIface {
    public String getSpeech(int act, int scene, int speech);
}

/*
 * PlayClient.java
 *
 * Version History
 * 1.0    22/2/99         Steve
 */

import UK.co.ansa.flexinet.core.naming.Tagged;
import UK.co.ansa.flexinet.trivtrader.FNetTrader;
import UK.ac.uwe.ics.followme.trader.TraderAdmin;
import UK.ac.uwe.ics.followme.trader.Trader;
import UK.ac.uwe.ics.followme.trader.TraderException;
import UK.ac.uwe.ics.followme.trader.OfferInfo;
import UK.co.ansa.flexinet.test.FNetTest;

public class PlayClient {

    public static void main(String[] args) {
  try {
      String context = args[0];
      String playName = args[1];
      int act = Integer.parseInt(args[2]);
      int scene = Integer.parseInt(args[3]);
      int speech = Integer.parseInt(args[4]);

      FNetTrader fnt = FNetTest.getTrader();
      if (fnt == null) throw new Exception("couldn’t find the TrivTrader");

      TraderAdmin admin = (TraderAdmin) fnt.get("FollowMeTraderAdmin");
      if (admin == null) throw new Exception("FollowMeTraderAdmin not resolved with
TrivTrader");

      Trader trader = (Trader) fnt.get("FollowMeTrader");
      if (admin == null) throw new Exception("FollowMeTrader not resolved with TrivTrader");
```

```
        // bind to the service
        OfferInfo[] offers ;
        offers = trader.importOffers("/"+context, "name=='"+playName+"'", null);
        for (int i=0 ; i<offers.length;i++) {
          Object service = offers[i].getTagged().iface() ;
          if (service instanceof PlayServerIface) {
              // service interaction
              System.out.println(((PlayServerIface) service).getSpeech(act,scene,speech));
          }
        }
    } catch (Exception error) {
        error.printStackTrace(System.err);
    } catch (TraderException error) {
        error.printStackTrace(System.err);
    }
      }

}
```

7.11 Appendix K - Type checking against a DTD

```
<MISSION>
  <?RULE UK.ac.uwe.ics.followme.scripting.Script implements SCRIPT?>
  <?RULE UK.ac.uwe.ics.followme.scripting.si.DTD implements DTD?>
  <?RULE UK.ac.uwe.ics.followme.scripting.si.XML implements PLAY?>

  <DTD name="play"><![CDATA[
    <!DOCTYPE PLAY [
      <!ATTLIST PLAY name CDATA #REQUIRED>
      <!ENTITY amp "&#38;">
      <!ELEMENT PLAY     (TITLE, FM, PERSONAE, SCNDESCR, PLAYSUBT, INDUCT?, PROLOGUE?, ACT+,
EPILOGUE?)>
      <!ELEMENT TITLE    (#PCDATA)>
      <!ELEMENT FM       (P+)>
      <!ELEMENT P        (#PCDATA)>
      <!ELEMENT PERSONAE (TITLE, (PERSONA | PGROUP)+)>
      <!ELEMENT PGROUP   (PERSONA+, GRPDESCR)>
      <!ELEMENT PERSONA  (#PCDATA)>
      <!ELEMENT GRPDESCR (#PCDATA)>
      <!ELEMENT SCNDESCR (#PCDATA)>
      <!ELEMENT PLAYSUBT (#PCDATA)>
      <!ELEMENT INDUCT   (TITLE, SUBTITLE*, (SCENE+|(SPEECH|STAGEDIR|SUBHEAD)+))>
      <!ELEMENT ACT      (TITLE, SUBTITLE*, PROLOGUE?, SCENE+, EPILOGUE?)>
      <!ELEMENT SCENE    (TITLE, SUBTITLE*, (SPEECH | STAGEDIR | SUBHEAD)+)>
      <!ELEMENT PROLOGUE (TITLE, SUBTITLE*, (STAGEDIR | SPEECH)+)>
      <!ELEMENT EPILOGUE (TITLE, SUBTITLE*, (STAGEDIR | SPEECH)+)>
      <!ELEMENT SPEECH   (SPEAKER+, (LINE | STAGEDIR | SUBHEAD)+)>
      <!ELEMENT SPEAKER  (#PCDATA)>
      <!ELEMENT LINE     (#PCDATA | STAGEDIR)*>
      <!ELEMENT STAGEDIR (#PCDATA)>
      <!ELEMENT SUBTITLE (#PCDATA)>
      <!ELEMENT SUBHEAD  (#PCDATA)>
    ]>
  ]]></DTD>

  <PLAY name="richard">
    <TITLE>The Tragedy of Richard the Third</TITLE>
    <FM>
      <P>Text placed in the public domain by Moby Lexical Tools, 1992.</P>
      <P>SGML markup by Jon Bosak, 1992-1994.</P>
      <P>XML version by Jon Bosak, 1996-1998.</P>
      <P>This work may be freely copied and distributed worldwide.</P>
    </FM>
    <PERSONAE>
      <TITLE>Dramatis Personae</TITLE>
      <PERSONA>RICHARD, Duke of Gloucester, afterwards King Richard III.</PERSONA>
    </PERSONAE>
    <SCNDESCR>SCENE  England.</SCNDESCR>
    <PLAYSUBT>KING RICHARD III</PLAYSUBT>
    <ACT><TITLE>ACT I</TITLE>
      <SCENE><TITLE>SCENE I.  London. A street.</TITLE>
        <STAGEDIR>Enter GLOUCESTER, solus</STAGEDIR>

        <SPEECH>
          <SPEAKER>GLOUCESTER</SPEAKER>
          <LINE>Now is the winter of our discontent</LINE>
          <LINE>Made glorious summer by this sun of York;</LINE>
          <LINE>And all the clouds that lour'd upon our house</LINE>
          <LINE>In the deep bosom of the ocean buried.</LINE>
          <LINE>Now are our brows bound with victorious wreaths;</LINE>
          <LINE>Our bruised arms hung up for monuments;</LINE>
          <LINE>Our stern alarums changed to merry meetings,</LINE>
          <LINE>Our dreadful marches to delightful measures.</LINE>
          <LINE>Grim-visaged war hath smooth’d his wrinkled front;</LINE>
          <LINE>And now, instead of mounting barded steeds</LINE>
          <LINE>To fright the souls of fearful adversaries,</LINE>
          <LINE>He capers nimbly in a lady’s chamber</LINE>
          <LINE>To the lascivious pleasing of a lute.</LINE>
          <LINE>But I, that am not shaped for sportive tricks,</LINE>
          <LINE>Nor made to court an amorous looking-glass;</LINE>
          <LINE>I, that am rudely stamp’d, and want love’s majesty</LINE>
          <LINE>To strut before a wanton ambling nymph;</LINE>
          <LINE>I, that am curtail’d of this fair proportion,</LINE>
          <LINE>Cheated of feature by dissembling nature,</LINE>
          <LINE>Deformed, unfinish’d, sent before my time</LINE>
          <LINE>Into this breathing world, scarce half made up,</LINE>
          <LINE>And that so lamely and unfashionable</LINE>
          <LINE>That dogs bark at me as I halt by them;</LINE>
          <LINE>Why, I, in this weak piping time of peace,</LINE>
          <LINE>Have no delight to pass away the time,</LINE>
          <LINE>Unless to spy my shadow in the sun</LINE>
          <LINE>And descant on mine own deformity:</LINE>
          <LINE>And therefore, since I cannot prove a lover,</LINE>
          <LINE>To entertain these fair well-spoken days,</LINE>
          <LINE>I am determined to prove a villain</LINE>
          <LINE>And hate the idle pleasures of these days.</LINE>
          <LINE>Plots have I laid, inductions dangerous,</LINE>
          <LINE>By drunken prophecies, libels and dreams,</LINE>
          <LINE>To set my brother Clarence and the king</LINE>
          <LINE>In deadly hate the one against the other:</LINE>
          <LINE>And if King Edward be as true and just</LINE>
          <LINE>As I am subtle, false and treacherous,</LINE>
          <LINE>This day should Clarence closely be mew’d up,</LINE>
          <LINE>About a prophecy, which says that ’G’</LINE>
          <LINE>Of Edward’s heirs the murderer shall be.</LINE>
          <LINE>Dive, thoughts, down to my soul: here</LINE>
          <LINE>Clarence comes.</LINE>
          <LINE>Brother, good day; what means this armed guard</LINE>
          <LINE>That waits upon your grace?</LINE>
        </SPEECH>
        <STAGEDIR>Exit</STAGEDIR>
      </SCENE>
    </ACT>
  </PLAY>

  <SCRIPT><![CDATA[
    console.writeln(richard) ;
    console.writeln(play.validate(richard));
  ]]></SCRIPT>
</MISSION>
```