JACK™ Intelligent Agents
Agent Practicals

Copyright

Copyright © 2012, Agent Oriented Software Pty. Ltd.

All rights reserved.

No part of this document may be reproduced, transferred, sold, or otherwise disposed of, without the written permission of the owner.

US Government Restricted Rights

The JACK™ Modules and relevant Software Material have been developed entirely at private expense and are accordingly provided with RESTRICTED RIGHTS. Use, duplication, or disclosure by Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of DFARS 252.227-7013 or subparagraph (c)(1) and (2) of the Commercial Computer Software Restricted Rights and 48 CFR 52.2270-19, as applicable.

Trademarks

All the trademarks mentioned in this document are the property of their respective owners.

Publisher Information

Agent Oriented Software Pty. Ltd.
P.O. Box 639,
Carlton South, Victoria, 3053
AUSTRALIA

Phone:	+61 3 9349 5055
Fax:	+61 3 9349 5088
Web:	http://www.agent-software.com

If you find any errors in this document or would like to suggest improvements, please let us know.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

The JACK™ documentation set includes the following manuals and practicals:

Document	Description
Agent Manual	Describes the JACK programming language and infrastructure. JACK can be used to develop applications involving BDI agents.
Teams Manual	Describes the JACK Teams programming language extensions. JACK Teams can be used to develop applications that involve coordinated activity among teams of agents.
Development Environment Manual	Describes how to use the JACK Development Environment (JDE). The JDE is a graphical development environment that can be used to develop JACK agent and team-based applications.
JACOB Manual	Describes how to use JACOB. JACOB is an object modelling language that can be used for inter-process transport and object initialisation.
WebBot Manual	Describes how to use the JACK WebBot to develop JACK enabled web applications.
Design Tool Manual	Describes how to use the Design Tool to design and build an application within the JACK Development Environment.
Graphical Plan Editor Manual	Describes how to use the Graphical Plan Editor to develop graphical plans within the JACK Development Environment.
JACK Sim Manual	Describes how to use the JACK Sim framework for building and running repeatable agent simulations.
Tracing and Logging Manual	Describes the tracing and logging tools available with JACK.
Agent Practicals	A set of practicals designed to introduce the basic concepts involved in JACK programming.
Teams Practicals	A set of practicals designed to introduce the basic concepts involved in Teams programming.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

Table of Contents

Practical 1 Introduction to JACK 9

Introductory notes 9

JACK 9

The JACK Agent Language 11

JACK execution 11

The Agent class 13

Agent template 13

Agent base members 13

Agent base methods. 14

An example. 15

The Event class 16

Normal events 16

BDI events 16

Event template 17

Event base members 18

Event methods 18

An example. 19

The Plan class 19

Plan template 20

Plan base members 21

Plan base methods. 21

An example. 21

Building our example 21

The Capability class 22

Capability template 22

Capability methods 22

An example. 23

The Beliefset class 23

Exercise 1. 24

Introduction. 24

Instructions 24

Exercise 2. 28

Introduction. 28

Instructions: 28

Questions 30

Exercise 3. 32

Introduction. 32

Instructions 33

Questions 35

Copyright © 2012, Agent Oriented Software Pty. Ltd.

Exercise 4. 36
Introduction. 36
Instructions 36
Exercise 5. 40
Introduction. 40
Instructions 40
Questions 41
Exercise 6. 42
Introduction. 42
Instructions 42
Exercise 7. 47
Introduction. 47
Instructions 47
Questions 52
Exercise 8. 53
Introduction. 53
Instructions 53
Questions 57
Exercise 9. 58
Introduction. 58
Instructions 58
Questions 58
Exercise 10. 59
Introduction. 59
Instructions 61
Exercise 11. 62
Introduction. 62
Instructions 63
Practical 2 JACK Beliefset Relations 69
Introductory notes 69
JACK beliefs 69
JACK beliefset definition. 71
An example. 72
Exercise 1. 75
Introduction. 75
Instructions 75
Exercise 2. 78
Introduction. 78
Instructions 78
Questions 79
Exercise 3. 80

Copyright © 2012, Agent Oriented Software Pty. Ltd.

Introduction. 80
Instructions 80
Exercise 4. 81
Introduction. 81
Instructions 81
Exercise 5. 83
Introduction. 83
Instructions 83
Practical 1 Solutions. 85
Program solutions 85
Answers to questions 85
Exercise 2. 85
Exercise 3. 85
Exercise 5. 85
Exercise 7. 85
Exercise 8. 85
Exercise 9. 85
Practical 2 Solutions. 87
Answers to questions in introductory notes 87
Program solutions 87
Answers to questions 88
Exercise 2. 88

Copyright © 2012, Agent Oriented Software Pty. Ltd.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

Practical 1 Introduction to JACK

Practical 1 provides an introduction to JACK agents, plans, events and capabilities. The notes are based on the material in the *JACK™ Intelligent Agents Agent Manual* and provide a summary of the features that you need to be familiar with to complete the programming exercises. For more details refer to the *Agent Manual*. When the prerequisite information has been covered for a particular exercise, it is indicated explicitly in the notes. In some cases additional information is provided in the introduction for the exercise. It is assumed that you are familiar with Java and are able to develop and run Java applications in your computing environment.

The Practical exercises will be developed using the JACK Development Environment. It is assumed that the reader is familiar with the overall structure and operation of the JDE. If this is not the case then at least Chapters 1 and 2 of the *JACK™ Intelligent Agents Development Environment Manual* should be read before commencing the exercises. No additional material regarding the JDE is provided in these notes.

Introductory notes

JACK

JACK™ Intelligent Agents (JACK) is an agent-oriented development environment built on top of and fully integrated with the Java programming language. JACK consists of the key components described below.

The JACK Agent Language

The JACK Agent Language (JAL) is a programming language that can be used to develop agent-based systems. JAL is a 'super-set' of Java – encompassing the full Java syntax while extending it with agent-oriented constructs.

The JACK Agent compiler

The JACK Agent Compiler pre-processes JAL source files and converts them into Java. This Java source code can then be compiled into Java virtual machine code to run on the target system.

The JACK Agent kernel

The JACK Agent kernel is the runtime engine for programs written in JAL. It provides a set of classes that give JAL programs their agent-oriented functionality. Most of these classes run behind the scenes and implement the underlying infrastructure and functionality that agents require. Others are used explicitly in JAL programs. They are inherited from and supplemented with callbacks as required to provide agents with their own unique functionality.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

The JACK Development Environment

The JDE is a cross-platform graphical development environment that can be used to develop JACK agent applications. In addition to providing support for code generation, tools are provided to support the design process and the tracing of agent execution.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

The JACK Agent Language

JAL extends Java by:

- Providing new base classes, interfaces and methods.
- Extending Java syntax to support the new classes, definitions and statements.
- Providing semantic extensions that change the *execution engine* to support an agent-oriented execution model.

Class-level constructs

`Agent`, `Event`, `Plan`, `Beliefset`, `View` and `Capability` class-level constructs are provided in JACK. Each of these are implemented as Java classes.

To create your own agent, the `Agent` class is extended and the particular details required for your specific agent are included. A similar approach is used to create your own events, plans, beliefsets and capabilities.

JACK declarations

These provide a set of statements that define properties of a JAL type and declare relationships between the classes above. They are preceded by a `#` symbol.

JACK reasoning method statements (@-statements)

Reasoning statements are JAL statements that can only appear in reasoning methods. Reasoning methods are found inside a plan. Reasoning method statements are preceded by an @ symbol.

Semantic extensions

JAL provides semantic extensions that support the *Belief Desire Intention (BDI)* execution model.

JACK execution

When an agent is instantiated in a system, it will wait until it is given a goal or it experiences an event to which it must respond. When it receives an event (or goal), the agent initiates activity to handle the event. If it does not believe that the goal or event has already been handled, it will look for the appropriate plan(s) to handle it. The agent then executes the plan or plans depending on the event type. The handling of the event may be synchronous or asynchronous relative to the posting. The plan execution may involve interaction with an agent's beliefset relations or other Java data structures. The plan being executed can in turn initiate other subtasks, which may in turn initiate further subtasks (and so on). Plans can succeed or fail. Under certain circumstances, if the plan fails, the agent may try another plan.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

Within a single process you can have multiple agents and each agent potentially has multiple *task queues*. A task queue is generated when an asynchronous event is received by the agent (either from itself or from another agent). The task queue contains the processing steps (or tasks) that are required for the agent to handle the event – these steps are specified in a plan. Task execution can result in further event posting. If the event is posted synchronously, the resulting tasks are added to the head of the task queue that generated the event. If the event is posted asynchronously, a new task queue will be generated.

The JACK kernel is responsible for giving each agent a 'turn'. Within an agent, a task manager is responsible for cycling through the task queues (exactly how will depend on the task manager being used by the agent). This is all managed by the JACK kernel within the JACK thread of execution. Note that this does not prevent other threads in the Java program from calling agent methods. You will see examples where an agent method is invoked from the Java main thread in the exercises.

Of course, you can also have agents in different processes and on different machines communicating with one another.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

The Agent class

The `Agent` class embodies the functionality associated with a JACK intelligent agent. To define your own agents, the JACK `Agent` class is extended by adding members and methods that are applicable to your agent's specific problem domain.

Agent template

In a file called `AgentType.agent`:

```
agent AgentType extends Agent {implements interface}
{
   // JAL declaration statements - the following declarations may be
   // used in an agent definition (when required).

   #{private,agent,global} data Type Name (arglist);

   // The agent handles events of type EventType.
   #handles event EventType;

   // The agent uses a plan of PlanType.
   #uses plan PlanType;

   // The round robin task manager is to be used - there are others.
   #uses taskManager SimpleRRTaskManager(steps);

   // The agent posts events of type EventType to itself.
   #posts event EventType reference;

   // The agent sends events of type EventType to other agents.
   #sends event EventType reference;

   // The agent has a capability of type CapabilityType.
   #has capability CapabilityType reference;

   // Data members (Java data structures).

   // Constructor method.
   AgentType(arglist)
   {
     super("agent name");
      :
      :
   }

   // Java methods that implement agent functionality.
   // (These may be called from within the agent's plans.)
}
```

Agent base members

`Timer timer`

Specifies which clock the agent uses to measure the passage of time.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

Agent base methods

`Constructor`

To construct an agent, follow the normal convention for constructors used in Java. JACK agents require a name (of type `String`).

`finish()`

Used to terminate an agent. It causes all event processing within the agent to be terminated immediately, and removes the agent from the JACK runtime network.

`postEvent(EventName)`

This allows an agent to post an event to itself. The event is handled asynchronously by the agent. The parameter is an instance of an event. As event posting methods are responsible for the creation of an instance of an event, `postEvent()` is often invoked with an argument that involves the event's posting method as shown below.

```
postEvent(eventRef.eventPostingMethod(args))
```

Note that `eventRef` must have been declared to be an event that is handled by the agent (`#handles event EventType eventRef;`), and the event must also have a corresponding posting method. The `EventName` parameter that is found in the methods described below is often replaced by an argument that involves the event's posting method.

`postEventAndWait(EventName)`

This is similar to `postEvent()` except that it is posted synchronously. The event is still executed as a separate task, but the calling method must wait until this task has been completed before continuing.

Note: Unlike an agent's other base methods, `postEventAndWait()` must not be called from any of an agent's tasks as it will block the agent. It can only be used from methods used by normal Java programs or other Java threads that are integrated with your JACK application. While the agent handles this event, the calling thread must wait until the agent returns its result. You will be using `postEventAndWait()` in the exercises.

`send(name, EventName)`

This method is used to send messages/events to other agents. The first parameter (`name`) is of type `String` and is the name of the destination agent. The second parameter is the message event to be sent to the destination agent. Only certain event types can be used for inter-agent communication. This will be discussed later in this document.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

`reply(receivedMessageEvent, SendEventName)`

`reply()` is used to send a message back to an agent from which a previous message has been received. `reply()` does not specify the destination agent – this information is contained in the message that it received.

The reply message arrives as a data object on the reply queue of the original message event. This means that the message event that is sent back using `reply` does not trigger a new task or plan.

`name()`

This method can be used to retrieve the agent's name as a `String`.

An example

In `MyAgent.agent`:

```
agent MyAgent extends Agent
{
  #handles event MyEvent;
  #uses plan MyPlan;
  #posts event MyEvent myEventRef;

  MyAgent(String name)
  {
    super(name);
  }

  public void method1(String exampleMessage)
  {
    // Java code to do something useful

    // In this example, we illustrate how to post an event from
    // within an agent.
    // The posting method will be discussed in the section on events.
    // In this case, the posting method takes a single argument of
    // type String.
    // Note that myEventRef has been declared earlier in the agent.

    postEvent(myEventRef.myPostingMethod(exampleMessage));
  }
}
```

Note: You may attempt exercise 1 now.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

The Event class

Events are the originators of all activity within JACK. There are a number of Event classes in JACK. They can be categorised into the two broad categories of normal events and BDI events.

Normal events

Normal events correspond to events in conventional event-driven programming. They are transitory and initiate a single immediate response. On receipt of the event, the agent selects an appropriate plan which either succeeds or fails. If the chosen plan fails, the agent does not try another plan.The normal event classes are:

Event Type	Description
`Event`	Base class for all normal events. Can only be posted internally.
`MessageEvent`	Can be used to send events between agents.

BDI events

These are used to represent a change in belief or circumstance that give the agent a sense of purpose. The agent desires not to react to information, but rather to achieve something. By default, all BDI agents can involve meta-level reasoning (i.e. reasoning about plan selection). Depending on the type of BDI event (and the event's behaviour attributes) the agent may try alternative plans and may even perform recalculation of the *applicable plan set* before selecting another plan to try. The BDI event classes are described in the following table.

Event Type	Description
`BDIFactEvent`	This type of event can only arise internally. By default, on receipt of a `BDIFactEvent`, the agent can perform meta-level reasoning, but it does not allow reconsideration of alternative plans if the plan fails.
`BDIMessageEvent`	This type of event can be used to send BDI events between agents. By default, on receipt of a `BDIMessageEvent`, the agent can perform meta-level reasoning, but it does not allow reconsideration of alternative plans if the plan fails.
`BDIGoalEvent`	This type of event represents a goal or objective that an agent wishes to achieve. It offers all the BDI features – meta-level reasoning, recalculation of the applicable plan set and plan re-selection if a plan fails. It can originate from within an agent using `@post`, `@subtask` etc. In addition, it can originate as a result of executing `@achieve`, `@insist`, `@test` and `@determine` statements.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

Note that there are two other event classes to consider.

Event Type	Description
`InferenceGoalEvent`	This is an event that uses `Rule behavior` which extends `BDI behavior` by processing all applicable plans.
`PlanChoice`	This is the mechanism the agent uses to perform meta-level reasoning.

The brief descriptions given above of the BDI events are of the default behaviours that occur when the event arises. This default behaviour can be modified by setting the behaviour attributes of an event using `#set behavior` statements in the event to set the values of particular attributes. `InferenceGoalEvent`s can also be customised by setting behaviour attributes. Details of the attributes can be found in the *Agent Manual*.

Event template

To define your own events, extend the appropriate JACK event class. Then add any members required as part of the event structure. These members can then be used to convey information to the agent when it receives the event. It is also necessary to specify at least one posting method for the event.

We will illustrate the event template using the base class `Event`. Note that to define an event to extend one of the other event types described earlier, you would replace `Event` with the specific event type required: i.e. `MessageEvent`, `BDIFactEvent`, `BDIMessageEvent`, `BDIGoalEvent`, `InferenceGoalEvent` or `PlanChoice`.

In a file called `EventType.event`:

```
event EventType extends Event
{
  // Any members required as part of the event structure.

  // Any declarations required to give the agent access to data or
  // beliefsets within the enclosing agent or capability.

  #uses data DataType data_name;

  // Any #posted when declarations required so that the event
  // will be posted automatically when certain belief states arise.

  #posted when { condition }

  // Declarations specifying how the event is posted within an agent.
  // You can have as many posting methods as you require.

  #posted as postingMethodName(parameter list)
  {
    // method body
  }
}
```


Copyright © 2012, Agent Oriented Software Pty. Ltd.

A posting method is executed whenever an instance of the event needs to be created. The posting method describes everything that the agent needs to do to construct an instance of the event.

Event base members

The actual base members will depend on the specific event type. They are:

Member	Description
`String from`	This member contains the address of the agent from which the event was sent. It is only present in the events that are used for inter-agent communication e.g. MessageEvents.
`String message`	This member is found in `MessageEvent` and `BDIMessageEvent` events. The information in message is used in the interaction diagram display.
`String mode`	This member is only found in BDIGoalEvents. It indicates whether the event arose from `@achieve`, `@insist`, `@test` or `@determine`.

Event methods

`Cursor replied()`

This method returns a cursor which can be `true` or `false` depending on whether or not an agent has received any replies to a given message event. A cursor is a special type provided by JACK. Cursors can be tested multiple times, and each time they are tested they may return a different truth value. They are discussed in more detail in the *Agent Manual*, but at this point it is sufficient to understand that this method can allow the agent to wait for a reply to a message before it continues with a given task.

`MessageEvent getReply()`

This complements `replied()`. It allows you to retrieve a reference to a message event that has been sent as a reply. If there are no replies it returns null. Again this method is only provided for message events.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

An example

In `MyEvent.event`

```
// Compare this with the earlier agent example.

event MyEvent extends BDIGoalEvent
{
  String text; // For example.

  #posted as
  myPostingMethod(String s)
  {
    text = s;
  }
}
```

The Plan class

A plan describes a sequence of actions that an agent can take when an event occurs. Each plan is capable of handling a single event. When an agent executes a plan, it starts by executing the plan's `body()` method – i.e. its top level reasoning method.

Reasoning methods are not the same as normal Java methods. Each statement in a reasoning method is treated as a logical statement that can either pass or fail. Unless the plan explicitly caters for the possibility of a statement's failure (through, for example, an `if-else` construct), failure of a statement causes a reasoning method to fail. If an agent reaches the end of a reasoning method, then the reasoning method has succeeded. If it reaches the end of the `body()` reasoning method, then the plan has succeeded.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

Plan template

The template below can be used for any plans that do not involve meta-level reasoning.

In a file called `PlanType.plan`

```
plan PlanType extends Plan
{
  // JAL declarations
  #handles event EventType eventref;

  // The following JAL declarations may be used in a plan definition
  // (as required).
  #posts event EventType eventref2;
  #sends event MessageEventType eventref3;
  #uses data Type ref;
  #reads data Type ref;
  #modifies data Type ref;
  #uses agent implementing InterfaceName agentref;
  #uses interface InterfaceName ref;

  #reasoning method methodName(parameter-list)
  {
    //Body of the reasoning method.
  }

  #reasoning method pass()
  {
    // Post-processing and clean up steps when the plan has succeeded.
  }

  #reasoning method fail()
  {
    // Post-processing and clean up steps when the plan has failed.
  }

  static boolean relevant (EventType eventref)
  {
    // Code to determine when the plan is relevant to eventref.
  }

  context()
  {
    // Logical condition to determine which plan instances are
    // applicable.
  }

  // The main reasoning method. All plans must have a body().
  body()
  {
    // The plan body - the actual steps performed when
    // the plan is executed.
  }
}
```

Note that reasoning methods can include special reasoning statements preceded by `@`. They are `@achieve`, `@determine`, `@insist`, `@maintain`, `@post`, `@reply`, `@send`, `@sleep`, `@subtask`, `@test` and `@waitFor`. These reasoning statements can only be used inside plans. For more details regarding the reasoning statements, refer to the *Agent Manual*.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

Plan base members

`Agent agent`

Identifies the agent to which the plan belongs

Plan base methods

Method	**Description**
`relevant()`	This was shown in the template and is discussed in the exercises.
`context()`	This was shown in the template and is discussed in the exercises.
`getInstanceInfo()`	This is used to retrieve information about the instance of the plan it is called on. It has been provided for use in meta-level reasoning plans. It is not discussed further in this practical.

An example

In a file `MyPlan.plan`

```
// Compare this with the example event and agent classes.
plan MyPlan extends Plan
{
  #handles event MyEvent me;

  body()
  {
    System.out.println(me.text);
  }
}
```

Building our example

We could test the example (which consists of `MyAgent.agent`, `MyEvent.event` and `MyPlan.plan`) with the following main program:

```
public class Program
{
  public static void main(String args[])
  {
    MyAgent agent1 = new MyAgent("agent1");
    agent1.method1("data to be posted");
  }
}
```

Note: You may attempt exercises 2 to 5 now.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

The Capability class

Capabilities represent functional aspects of an agent that can be 'plugged in' as required. Capabilities are built in a similar fashion to simple agents – constructing them involves declaring the JAL elements required. Events, beliefsets, plans, Java code and other capabilities can all be combined to make a capability.

Capability template

In a file called `CapabilityType.cap`:

```
capability CapabilityType extends Capability [implements Interface]
{
  // JAL declarations specifying the functionality associated with
  // the capability. The following declarations may be used in a
  // capability definition.
  #handles event EventType;
  #handles external {event} EventType;
  #posts event EventType {reference};
  #posts external event EventType {reference};
  #private data Type name({args});
  #exports data Type name({args});
  #imports data Type name();
  #uses plan PlanType;
  #has capability CapabilityType reference;

  // Other data members and method definitions.
}
```

Capability methods

`String toString()`

Capabilities do not have a name in the sense that agents do, but they can be referred to through the chain of reference names used in the `#has capability` statements. The reference name can be retrieved by calling the `toString()` method.

`void postEvent( Event event )`

The postEvent method is used to post events within capability code.

`Agent getAgent()`

If this method is called on a capability, it returns the containing agent.

`protected void autorun()`

This method can be overridden in order to provide some initialisation when the capability is actually brought into being.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

An example

In a file called `MyCapability.cap`:

```
capability MyCapability extends Capability
{
  #handles event MyEvent;
  #uses plan MyPlan;
}
```

The Beliefset class

JACK provides a `Beliefset` class which has been specifically designed to work within the agent-oriented paradigm. The JACK `Beliefset` class is described in detail in the notes provided with practical 2.

Note: You may now complete the practical 1 exercises 6 to 11.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

Exercise 1

Make a basic robot agent that contains a Java method that prints a message to indicate that the robot is painting a part.

Introduction

The aim of this exercise is to demonstrate how to build, compile and run a JACK program using the JACK Development Environment (JDE). Consequently, no agent-oriented programming concepts other than agent creation are involved.

Note: The practicals will make use of the design tool to produce design diagrams and to create initial skeleton type definitions for the application. The JDE supports multiple ways to add detail to the type definitions. However, to avoid confusion during the practicals we will always use the Edit as JACK File option to add type definition details.

Instructions

1. Make sure that `CLASSPATH` is set to include `jack.jar` and the root path for your application.

2. In a new directory (called `ex1`) start the JDE using the following command:

```
java -Xmx90m aos.main.Jack
```

3. Create a new project as follows:
   - Left-click on the File menu at the top left corner of the screen.
   - Left-click on New Project.
   - A New Project File dialog box will appear.
   - Name the project file `PaintRobot` and click on the New button. The project file suffix (`.prj`) will be added automatically.
   - The New Project File dialog box will close and return the user to the JDE screen.
   - In the top left corner of the browser window, the name of the new project will appear in red text.

4. We will now develop our application using the design tool. In this application we will develop the following types of diagrams:
   - A top level agent diagram which includes the details of messages sent/handled by an agent.
   - An agent-capability hierarchy diagram for each agent.
   - A Data-Event-Plan (DEP) diagram for each capability.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

Note that when we add new components to a design diagram, the skeleton for the type definition is automatically created and added to the browser. Links created between components on the design canvas result in the corresponding declarations being added to the type definitions in the browser. Removing a link from a design diagram removes the link from the underlying model and the change will be reflected in the browser. Removing a component from the design diagram does not remove it from the underlying model. It will still exist in the browser and can be dragged back on to the design canvas to view its relationships with other components in the diagram.

5. Create the top level agent diagram using the design tool.

   - To access the design tool, right-click on the Design Views folder in the JDE. A pop-up menu will appear.
   - Select Add New Design View from the menu. A pop-up dialog box will appear requesting a name for the design diagram.
   - Give the design a name (`Robot_AD`) and click on the Add New button.
   - The design tool canvas will appear. Left-click on the Open design palette button on the design tool menu bar.
   - Drag the Agent icon from the design tool palette to the design tool canvas.
   - Fill in the pop-up dialog box with details of the `Robot` agent. Make sure that the `Robot` agent is in a package called `robot`. Click on the Add New button. (The agent will now be displayed in the browser in the Agent Types folder inside the Agent Model.)
   - You can choose to be in either Selection mode or Link mode. To enter Selection mode, click the Change to selection mode button on the design tool tool bar (the Change to selection mode button displays an arrow icon). In Selection mode, you can click on entities on the canvas and change their location on the diagram. Selected entities or links can also be removed by first selecting them, then pressing the Delete selected objects button. The Delete selected objects button is the button displaying a cross icon. To enter Link mode, click on the Change to link mode button (the Change to link mode button displays a diagonal line). In Link mode you can make connections between the entities on your diagram. Note that the direction of a link is significant.

6. In the browser, edit the `Robot` agent by right-clicking on the robot agent and selecting Edit as JACK file. Add a Java method called `paintPart()` that contains the statement

```
System.out.println("Painting part now").
```

Close the robot agent file to ensure that there are no conflicts between editing the file in the JDE browser and as a JACK file. In the editor window of the robot agent file, click the Save button and then the Close button.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

7. Use the browser to add the main program. This is added in the Other Files folder at the top level of the browser hierarchy. Right-click on the Other Files folder and select Add New File from the pop-up menu. Add the code for the main program in the Edit File window which appears. Save the main program with the name `Program.java`. The following main program can be used to test the paint robot:

```
import robot.Robot;

public class Program {
   public static void main( String args[] ) {
      Robot robot1 = new Robot("robbie");
      robot1.paintPart();
      System.exit(0);
   }
}
```

8. Save the project. This can be achieved by selecting the Save Project option from the File menu in the top left hand corner of the JDE.

The JDE window should look similar to the following:

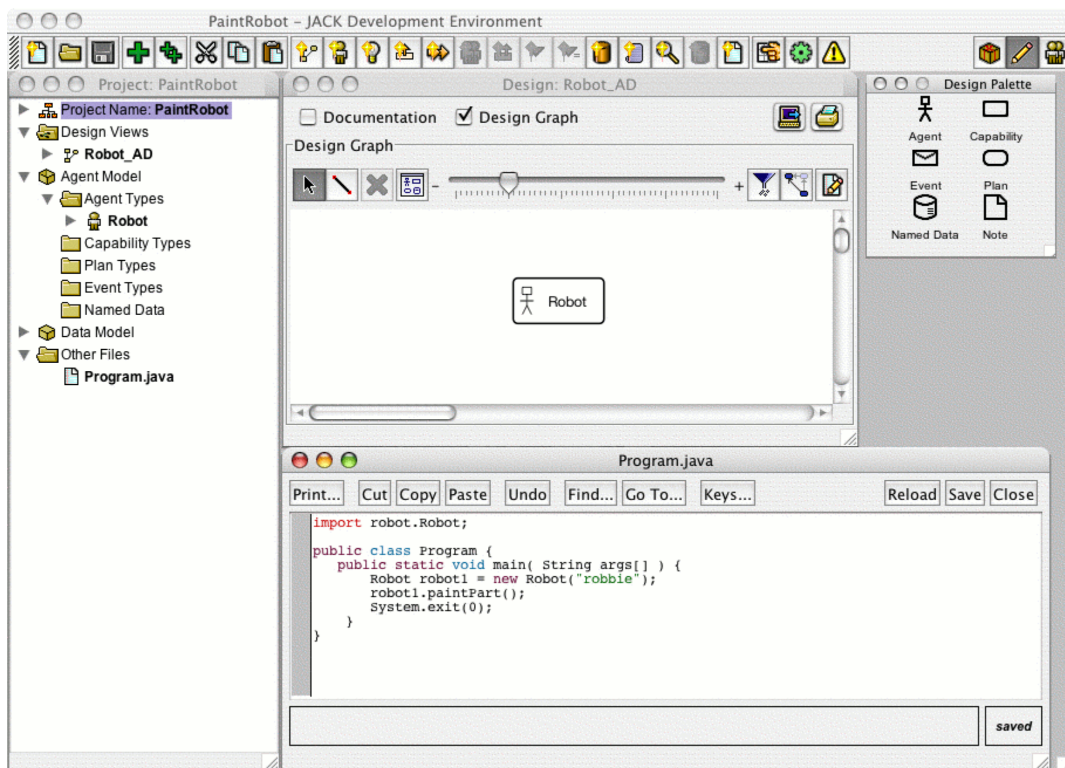

Figure 1: The JDE window with project browser, Robot_AD design, design palette and Program.java windows

Copyright © 2012, Agent Oriented Software Pty. Ltd.

9. Compile and run the program within the JDE.

   – Click on the Tools menu and select Compiler Utility from the drop-down menu.

   – Click on the Compile Application tab and click the Compile button. If there are any errors, they can be viewed in the Output/Errors tab in the Compiler Utility.

   – When you have successfully compiled the application, click on the Run Application tab in the Compiler Utility. Click on `Program.class` from the contents window and then click on the Select File button. The Run button can then be selected to start running the program.

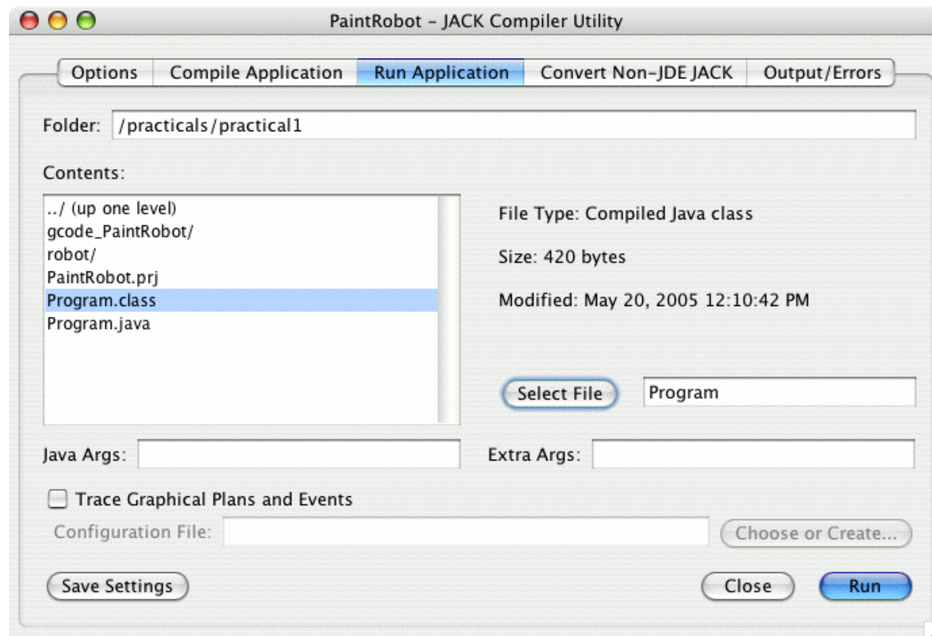

Figure 2: The Run Application tab of the Compiler Utility

Copyright © 2012, Agent Oriented Software Pty. Ltd.

Exercise 2

Extend the robot agent to use a JACK plan.

Introduction

In this exercise, the robot agent's `paintPart()` method will post a `Paint` event, thus allowing a plan to be chosen to respond to this event. At this stage we will only have one plan, `PaintSpecifiedCurrentColour`, which handles the `Paint` event. The plan will print a message to indicate that the robot is painting a part a particular colour. Note that the plan `PaintSpecifiedCurrentColour` would be more appropriately named `PaintSpecifiedColour` at this stage in the practical. However, this plan will be used to paint a part a specified colour which matches the robot's current paint colour in later exercises, and it was felt that it would be less confusing if the name was not changed.

As before, `paintPart()` is invoked from a Java `main()` method (and runs on the Java main thread) and is followed by a `System.exit(0)` call.

Plans and events are described in the *Introduction to JACK* notes. If necessary, read these before beginning the exercise.

Instructions:

1. Use the design tool to add a `Paint` event to the application.
   - If necessary, open the `Robot_AD` canvas by right-clicking on Robot_AD in the Design Views folder and selecting Edit "Robot_AD" from the pop-up menu.
   - Drag an event from the design palette onto the canvas.
   - Fill in the pop-up dialog box with details of the `Paint` event. Make sure that the `Paint` event is in the `robot` package.
   - Click on the Add New button. (The event will now be displayed in the browser in the Event Types folder inside the Agent Model.)

2. On the design canvas, create a `posts` link from the `Robot` agent to the `Paint` event. Observe that a declaration is added automatically by the JDE in the External Events folder of the `Robot` agent.

3. On the design canvas, create a `handles` link from the `Paint` event to the `Robot` agent.

4. Add a `PaintSpecifiedCurrentColour` plan to the `Robot_AD` canvas. The plan must also be in the `robot` package.

5. On the design canvas, create a `uses` link from the `Robot` agent to the `PaintSpecifiedCurrentColour` plan.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

6. On the design canvas, create a `handles` link from the `Paint` event to the `PaintSpecifiedCurrentColour` plan. Your design diagram should now be similar to the following diagram:

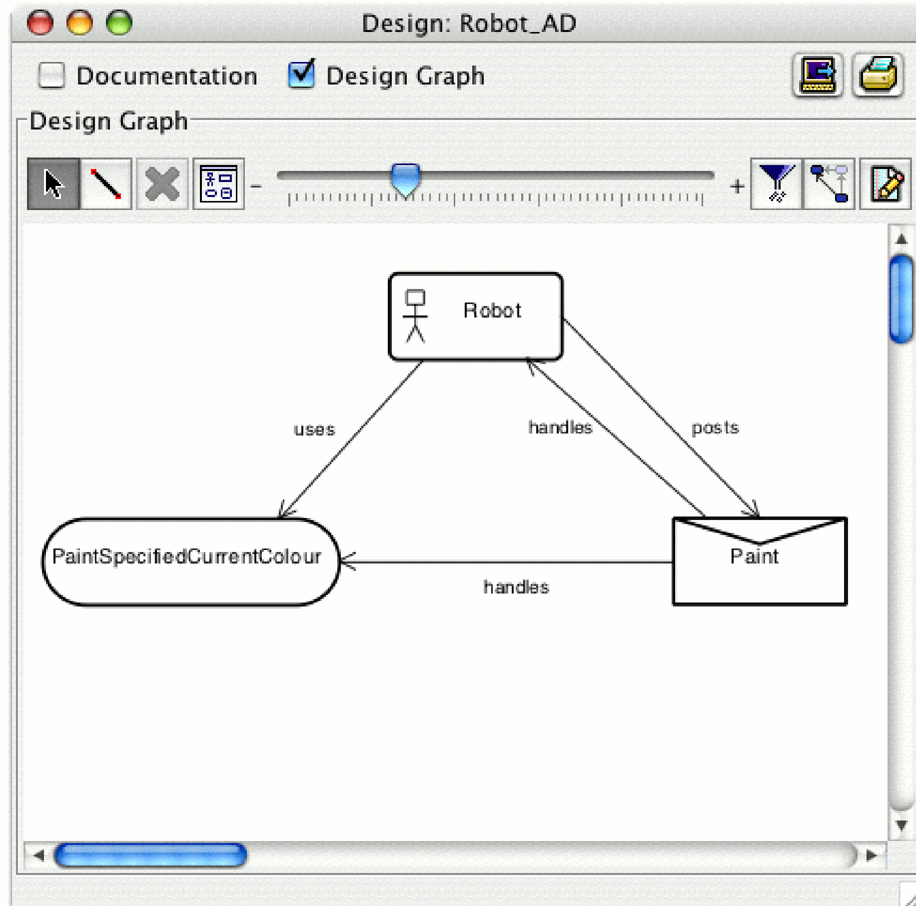

Figure 3: The `Robot_AD` design diagram with one plan, `PaintSpecifiedCurrentColour`

7. In the browser, right-click on the `Paint` event and select Edit as JACK file from the pop-up menu. Complete the `Paint` event by making it:

– Extend `BDIGoalEvent`;

– Contain a `String` data member(field) called `colour` (this is the information that will be carried with the event); and

– Include a posting method, `paint(String c)`. The parameter `c` is to be assigned to the event data member `colour` inside the posting method. The completed `Paint` event follows:

Copyright © 2012, Agent Oriented Software Pty. Ltd.

```
package robot;

public event Paint extends BDIGoalEvent {
   public String colour;

   #posted as
   paint(String c)
   {
      colour = c;
   }
}
```

Close the file to ensure that there are no conflicts between editing the file in the JDE browser and as a JACK file. In the editor window of the file, click the Save button and then the Close button.

8. Using the Edit as a JACK File option, edit the `Robot` agent and

- modify the `#posts event Paint` declaration so that it becomes:

  ```
  #posts event Paint pev;
  ```

- modify the `paintPart()` method in the `Robot` agent so that it uses the `postEventAndWait()` method to post a `Paint` event with the colour "white" as follows:

  ```
  postEventAndWait(pev.paint("white"));
  ```

Close the file to ensure that there are no conflicts between editing the file in the JDE browser and as a JACK file. In the editor window of the file, click the Save button and then the Close button.

9. Using the Edit as a JACK File option edit the `PaintSpecifiedCurrentColour` plan as follows:

- check that the reference to the `Paint` event handled by the plan is `ev`
- add the following statement to the plan's `body` reasoning method:

  ```
  System.out.println("Painting the part the requested colour: " +
                      ev.colour);
  ```

Close the file to ensure that there are no conflicts between editing the file in the JDE browser and as a JACK file. In the editor window of the file, click the Save button and then the Close button.

10. Save, compile and run the application.

Questions

1. What would happen if you used `postEvent()` inside `paintPart()` instead of `postEventAndWait()`? Try it!

Copyright © 2012, Agent Oriented Software Pty. Ltd.

If you did not observe any difference, add `@sleep(2)` before the print the statement in the `PaintSpecifiedCurrentColour` plan. Run the program using `postEventAndWait()` in the `paintPart()` method. Replace the `postEventAndWait` with `postEvent()` and run the program again. Explain your observations. Remove the `@sleep(2)` from the plan before you begin the next exercise.

Make sure you change `postEvent()` back into `postEventAndWait()` before you begin the next exercise.

2. `postEventAndWait()` should only be called from the Java main thread or from other Java threads that are external to JACK. While the agent handles this event, the calling thread is blocked and must wait until the agent returns its result. What problem would you envisage if `postEventAndWait()` was called from an agent task?

Copyright © 2012, Agent Oriented Software Pty. Ltd.

Exercise 3

Enable the robot agent to select between multiple plans through relevance.

Introduction

In this example, the agent has two plans that can handle a `Paint` event: `PaintAnyColour` and `PaintSpecifiedCurrentColour`.

The `PaintAnyColour` plan prints the string `"No colour specified. Painting the part."`

The `PaintSpecifiedCurrentColour` plan prints the string `"Painting part the requested colour:"` followed by the colour that was requested in the `Paint` event.

Whenever an event is posted and an agent begins a task to handle the event, the first thing that the agent must do is find an applicable plan to handle the event. Note that each plan is only capable of handling a single event type which is identified by the plan's `#handles event` declaration. It is possible (as in this example) that there is more than one plan capable of handling a particular event type. To decide which of the plans are applicable, JACK employs the following steps.

1. Identify the plans which handle the event type.
2. Use the `relevant()` method to check additional information regarding the event.
3. Use the `context()` method to check information stored as part of the agent's beliefs.
4. If there are still multiple plans left in the applicable plan set, additional means are used to select one of them. At this stage, we will only be considering *prominence* (or declaration order). For more details, refer to the *Agent Manual*.

We will be looking at the `relevant()` method in this example. The `PaintSpecifiedCurrentColour` plan will contain a `relevant()` method to ensure that the plan is only selected if there is a colour specified in the `colour` data member of the `Paint` event. If a plan does not specify a `relevant()` method, the plan is relevant for all instances of the event. The `relevant()` method takes the following form:

```
static boolean relevant(EventType eventref)
{
   // Code to determine when the plan is relevant to this event.
}
```

The next level of 'filtering' in plan selection (assuming the plan can handle the event and is relevant for a particular instance of the event) is the `context()` method. This is discussed in Exercise 4.

Note that the order of plan declarations within the agent (or capability) also has a bearing on plan selection. The order is called prominence, and the most prominent applicable plan will be selected first. This means that if `PaintAnyColour` is declared before

Copyright © 2012, Agent Oriented Software Pty. Ltd.

PaintSpecifiedCurrentColour, it will always be the plan selected to handle Paint events as it does not have a relevant() or a context() method associated with it.

Instructions

1. Use the design tool to add a new plan called PaintAnyColour to the Robot_AD design diagram.

- Add a link from the Robot agent to the new plan.
- Add a link from the Paint event to the new plan. Your design diagram should be similar to the following diagram:

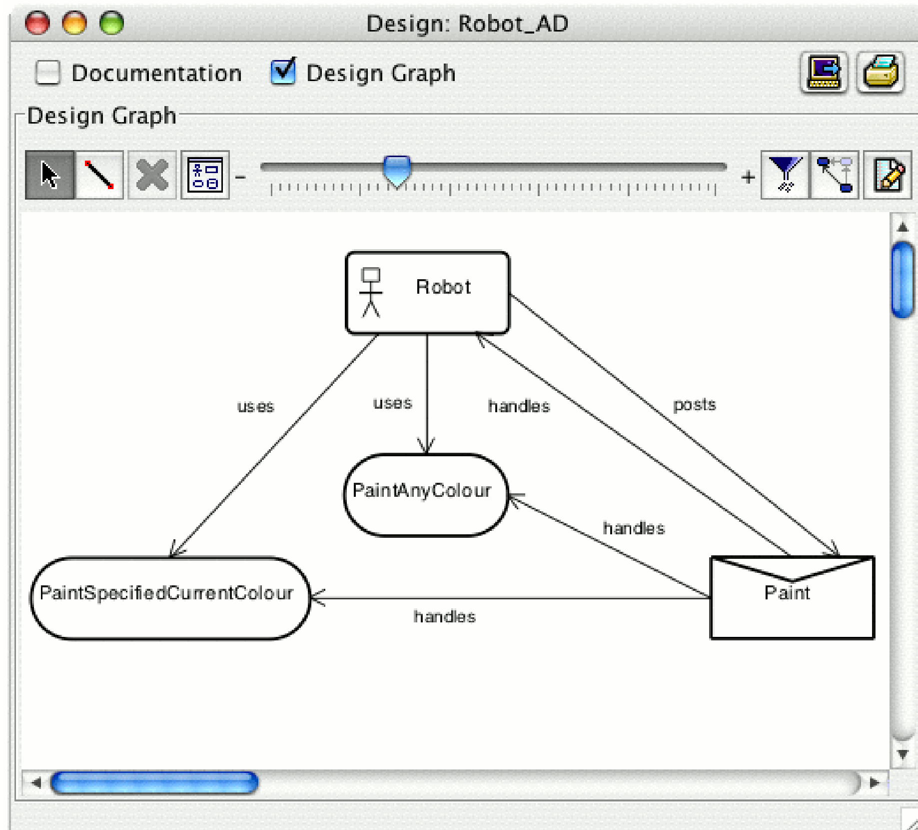

Figure 4: The Robot_AD design diagram with the PaintSpecifiedCurrentColour and PaintAnyColour plans

Copyright © 2012, Agent Oriented Software Pty. Ltd.

2. Use the Edit as a JACK File option to modify the `PaintSpecifiedCurrentColour` plan so that its `relevant()` method performs a test that recognises events with non-empty strings. For example,

```
static boolean relevant(Paint ev)
{
   return (ev.colour != null && ev.colour.length() > 0);
}
```

This plan will be used to paint a part the colour that was requested in the `colour` member of the `Paint` event. Note that if the robot is not currently painting with the requested colour an alternative plan will be required. We will not write that plan or test for that condition in this exercise – that is deferred until the next exercise.

Close the file to ensure that there are no conflicts between editing the file in the JDE browser and as a JACK file. In the editor window of the file, click the Save button and then the Close button.

3. Use Edit as a JACK File to edit the new plan to add the following print statement to its `body` reasoning method:

```
System.out.println("No specified colour. Painting the part");
```

This is the plan to be used when a `Paint` event does not include a specific colour request. The `colour` string will be null or empty. In this case the part will be painted with the colour being used by the robot at the time of the paint request.

Close the file to ensure that there are no conflicts between editing the file in the JDE browser and as a JACK file. In the editor window of the file, click the Save button and then the Close button.

4. The `#uses plan PaintAnyColour` declaration should appear after the `PaintSpecifiedCurrentColour` plan declaration. Check the order by editing the `Robot` agent. If necessary swap the declarations via the project browser or by editing the file as a JACK file.

If editing the file as a JACK file, save and close the file before continuing.

5. Edit the `Robot` and modify its `paintPart()` method so that it now takes a `String` argument (the colour) and passes it to the `paint` posting method.

If editing the file as a JACK file, save and close the file before continuing.

6. Edit the `main()` method in `Program.java`, so that the `paintPart()` method is invoked with the first program argument, if any, and null otherwise. For example:

```
robot1.paintPart( (args.length==0) ? null : args[0] );
```

Save and close the file to apply the changes before continuing.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

7. Save the project.

8. Compile and run the program a few times with and without command line arguments. The command line arguments are added in the Extra Args text box of the Compiler Utility's Run Application window.

9. Swap the plan declarations within the agent. Compile and run the program again (with and without command line arguments).

10. Swap the plan declarations back to normal before moving on to Exercise 4.

Questions

1. How is JACK choosing which plan to use when both plans are applicable?

Copyright © 2012, Agent Oriented Software Pty. Ltd.

Exercise 4

Enable the robot agent to select between multiple plans through *context*, as well as relevance.

Introduction

The `context()` method provides the next level of 'filtering' after `relevant()`. If a plan is relevant to a particular event, the `context()` method determines whether the plan is applicable given the agent's current knowledge. The `context()` method does not take any arguments and its body is always a single *JACK Agent Language logical expression*. (JAL logical expressions are composed of boolean members, logical members and beliefset cursor expressions which can, in general, bind to multiple values. Logical expressions and cursors are discussed in more detail in the *Agent Manual*.) When evaluating the `context()` method, the agent will consider all possible alternatives. Note that for every possible set of values that can satisfy the `context()` method, a separate instance of the plan will be generated and will be available for execution. This concept of multiple possible bindings and plan instances is illustrated in the introductory beliefset exercise found in Practical 2.

The `context()` method takes the following form:

```
context()
{
   // Logical condition to determine which plan instances are
   // applicable (in this example test the value of paintColour).
}
```

To illustrate the use of `context()` in a plan, the robot agent will have a `String` data member called `paintColour`, which stores the colour being used.

In this example we will also use

```
#uses interface Robot self;
```

within a plan. This statement gives the plan access to the members and methods in the `Robot` class interface. In this particular example, it will allow the plan to access the robot's `paintColour` member (`self.paintColour`).

Instructions

1. Use the design tool to add a new plan called `PaintSpecifiedNewColour` to the `Robot_AD` diagram. This plan is to deal with `Paint` events that request a specific colour, but which could not be handled by the `PaintSpecifiedCurrentColour` plan, as the colour requested is not the same as the agent's current `paintColour`. In the `PaintSpecifiedNewColour` plan, the agent's `paintColour` member must first be changed to the new colour. In addition, when the colour changes, it is necessary to give the part two coats of paint to ensure that no trace of the previous colour remains. Use the design tool to add the required links between the plan, the agent and the `Paint` event as follows:

- create a link from the `Paint` event to the plan

Copyright © 2012, Agent Oriented Software Pty. Ltd.

– create a link from the `Robot` agent to the plan

Your design diagram should be similar to the following diagram:

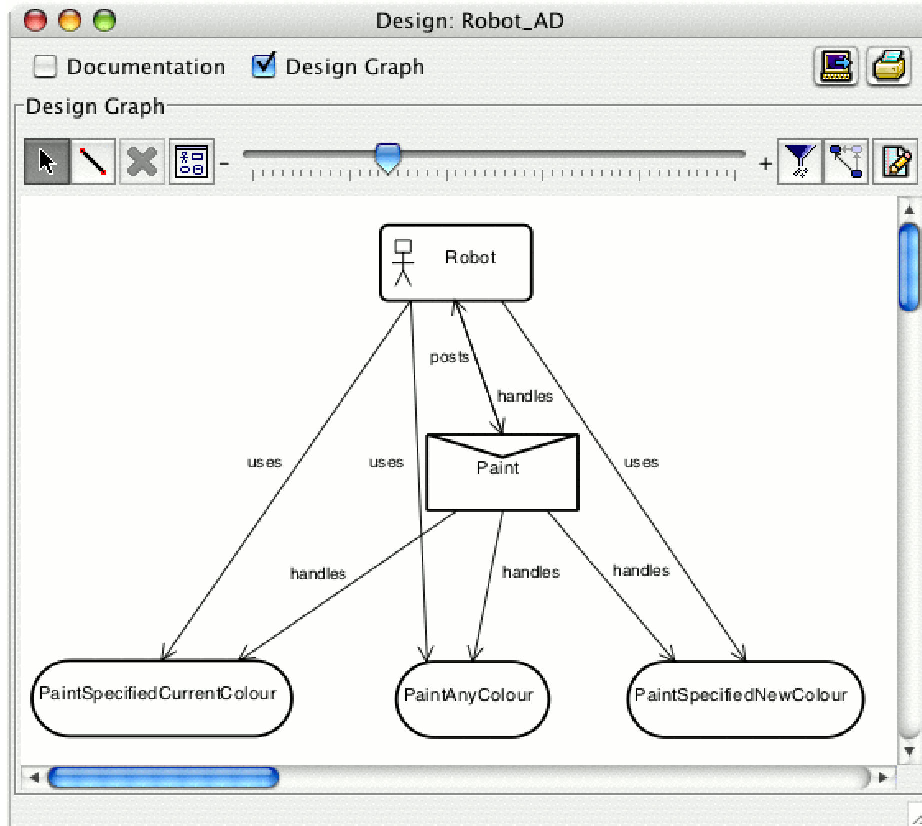

Figure 5: The `Robot_AD` design diagram with the `PaintSpecifiedCurrentColour`, `PaintAnyColour` and `PaintSpecifiedNewColour` plans

2. Use Edit as a JACK File to modify the `Robot` agent to:

– contain a `paintColour` member of type `String` with initial value `black` (this stores the colour currently being used by the robot); and

– contain a method `setColour(String colour)` that changes `paintColour` to `colour`.

Ensure that the plans are declared in the following order:

```
#uses plan PaintSpecifiedCurrentColour;
#uses plan PaintSpecifiedNewColour;
#uses plan PaintAnyColour;
```

Close the file to ensure that there are no conflicts between editing the file in the JDE browser and as a JACK file. In the editor window of the file, click the Save button and then the Close button.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

3. Use the Edit as a JACK File option to make the following changes to the new `PaintSpecifiedNewColour` plan:

   – check that it has the following declaration with `ev` used as the reference to the event:

   ```
   #handles event Paint ev;
   ```

   – modify the `relevant()` method to check that the `Paint` event's colour is a non-empty string;

   – add a `#uses interface Robot self` declaration so that it can access the robot's `setColour` method to change the colour;

   – invoke the `setColour` method to change the robot's `paintColour` to the colour requested by the `Paint` event. This must be invoked inside the plan's `body()` method before the 'painting' begins; and

   – print the string `"Painting part the requested colour: "+self.paintColour` using an appropriate print statement inside the `body()` method of the plan. This message should appear twice, indicating that the part received two coats of paint.

Close the file to ensure that there are no conflicts between editing the file in the JDE browser and as a JACK file. In the editor window of the file, click the Save button and then the Close button.

4. Use the Edit as a JACK File option to edit the `PaintSpecifiedCurrentColour` plan.

   – add a `#uses interface Robot self` declaration so that it can access the robot's `setColour` method to change the colour;

   – modify the `context()` method to test the agent's current `paintColour` as follows:

   ```
   context()
   {
      self.paintColour.equals(ev.colour);
   }
   ```

   – change the print statement in `body()` to

   ```
   System.out.println("Painting part the current colour "+
                             self.paintColour);
   ```

   This will make it easier to distinguish between the alternative plans in the program output.

Close the file to ensure that there are no conflicts between editing the file in the JDE browser and as a JACK file. In the editor window of the file, click the Save button and then the Close button.

5. Modify the `PaintAnyColour` plan so that it now prints the string `"No colour specified. Painting the part"` followed by the robot's current `paintColour`. To access the `paintColour` member inside the plan, add a `#uses interface Robot self` declaration at the beginning of the plan.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

If editing the file as a JACK file, save and close the file before continuing.

6. Modify the `main()` method in `Program.java` so that it no longer passes a command line argument to the `paintPart()` method but instead invokes the method several times as follows:

```
System.out.println("test with red");
robot1.paintPart("red");      // Should result in two coats.
System.out.println("test with no specified colour (null)");
robot1.paintPart(null);       // Should only be one coat -
                              // slightly different message.
System.out.println("test with green");
robot1.paintPart("green");    // Should result in two coats.
System.out.println("test with green again");
robot1.paintPart("green");    // Should only be one coat.
```

Save and close the file to apply the changes before continuing.

7. Save the project.

8. Compile and run the program. Check that the output is correct.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

Exercise 5

Illustrate the reposting of events when a plan fails.

Introduction

The way that an agent handles an event depends on the type of event. For example, when a normal event is received by an agent, the agent initiates a task to handle the event. This task involves selecting and executing the first plan that is both relevant and applicable to this event. With normal events the initial selection is the only plan that is executed. In the case of BDI events, the agent can apply more sophisticated reasoning to the plan selection, and can also attempt to achieve its goal using other applicable plans if a plan fails.

This exercise illustrates how a `BDIGoalEvent` will attempt every applicable plan until it succeeds. It will only fail when no more applicable plan instances remain to be tried. The `body()` method in a plan is the plan's main reasoning method. It is executed whenever a plan is executed. Reasoning methods do not have the same execution structure as ordinary Java methods. Each statement in a reasoning method is treated like a boolean expression, and each semicolon between statements like an AND connector. This means that if any statement fails, then the reasoning method terminates immediately and fails. In this exercise you can make the plan fail by adding the following statement in its `body()` method:

```
false;
```

Instructions

1. Modify the `PaintSpecifiedNewColour` plan so that it prints the painting message once and then fails (add `false;` after the first print statement). To illustrate that the plan actually stops at that point, add `System.out.println("After false statement");` after the `false` statement. This print statement should not be executed.

If editing the file as a JACK file, save and close the file before continuing.

2. Save the project.

3. Compile and run the program.

4. Add a `fail()` reasoning method that prints the string `"PaintSpecifiedNewColour plan failed"`.

If editing the file as a JACK file, save and close the file before continuing.

5. Save the project.

6. Compile and run the program.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

Questions

1. Which plan is now responsible for printing the second paint message that corresponds to the second coat of paint?

Copyright © 2012, Agent Oriented Software Pty. Ltd.

Exercise 6

Provide the robot agent with a `Painting` capability.

Introduction

The capability concept is a means of structuring reasoning elements of agents into 'clusters' that implement selected reasoning capabilities. This technique simplifies agent system design and allows code reuse and encapsulation of agent functionality. In this example, we encapsulate both the set of plans you have written so far and the `Paint` event into a `Painting` capability.

Capabilities are described in the *Introduction to JACK* notes. If necessary, read through the notes before beginning the exercise.

Instructions

1. Add a new design diagram called `Painting_DEP`. This will be the DEP diagram for a new `Painting` capability. Then

   - Drag a new capability from the design palette onto the new canvas. Call this new capability `Painting` and add it to the `robot` package.
   - Drag the three painting plans from the browser onto the `Painting_DEP` design diagram.
   - Drag the `Paint` event from the browser onto the `Painting_DEP` design diagram.
   - Create links from the `Painting` capability icon to each of the plans.
   - Create a `handles` link from the `Paint` event to the `Painting` capability.
   - The diagram becomes quite cluttered if we keep the capability on its DEP diagram. It is useful to have it on the diagram to make the links and to check the links. However, when this task is complete, the capability icon can be removed from the diagram without changing the links. This is achieved by using the design tool in Selection mode, selecting the capability icon on the canvas and then selecting the Remove selected objects from diagram button. Use this procedure to remove the capability from the `Painting_DEP` design diagram. Your `Painting_DEP` diagram should be similar to the following diagram.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

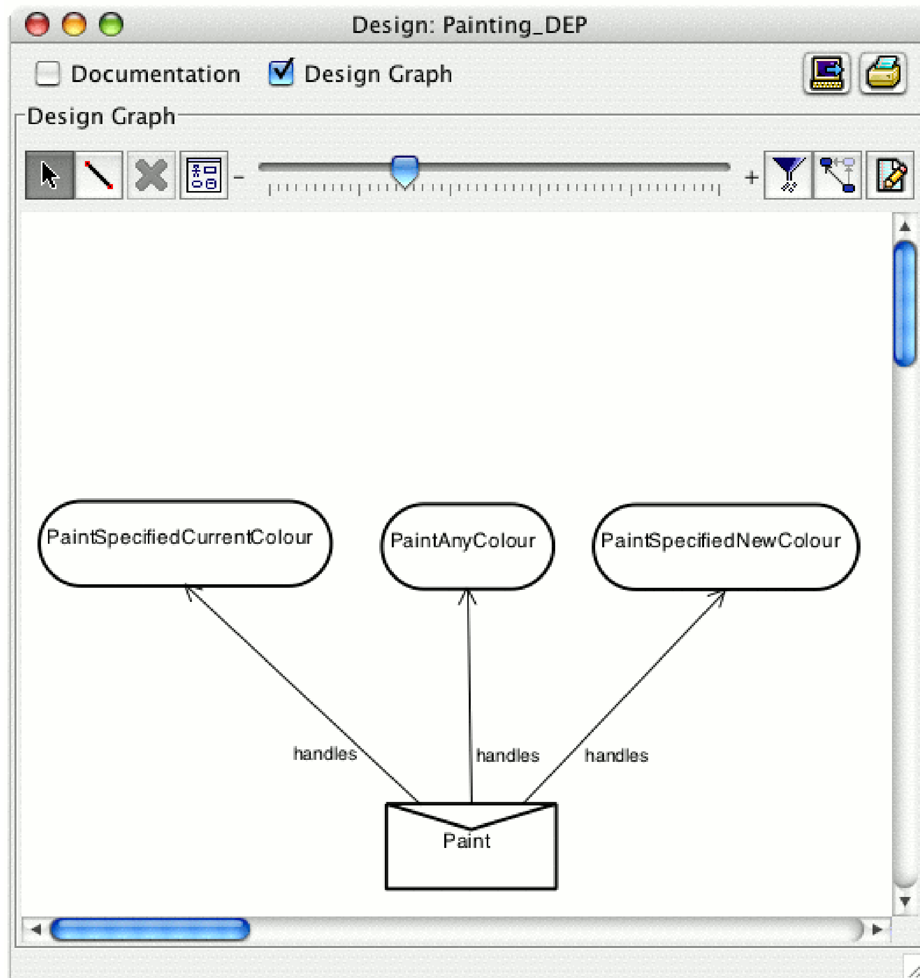

Figure 6: The `Painting_DEP` design diagram with the `PaintSpecifiedCurrentColour`, `PaintAnyColour` and `PaintSpecifiedNewColour` plans

2. Now that the plans and event are encapsulated in the new capability, we should remove the links from the agent to the plans and declare that the `Robot` agent has the `Painting` capability. This can be achieved by:

- deleting the `uses` links between the `Robot` agent and each of the painting plans on the `Robot_AD` design diagram. Take care that you do not delete the link that corresponds to the *#uses interface Robot self* declaration in each of the plans. An alternative mechanism for removing the links from the `Robot` to the plans is to edit the `Robot` type definition using the Edit as JACK File option.
- deleting the `handles` link between the `Paint` event and the `Robot` agent on the `Robot_AD` design diagram.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

- removing the components no longer required on the `Robot_AD` design diagram (i.e. remove the three painting plans from the design diagram). Your `Robot_AD` diagram should be similar to the following diagram:

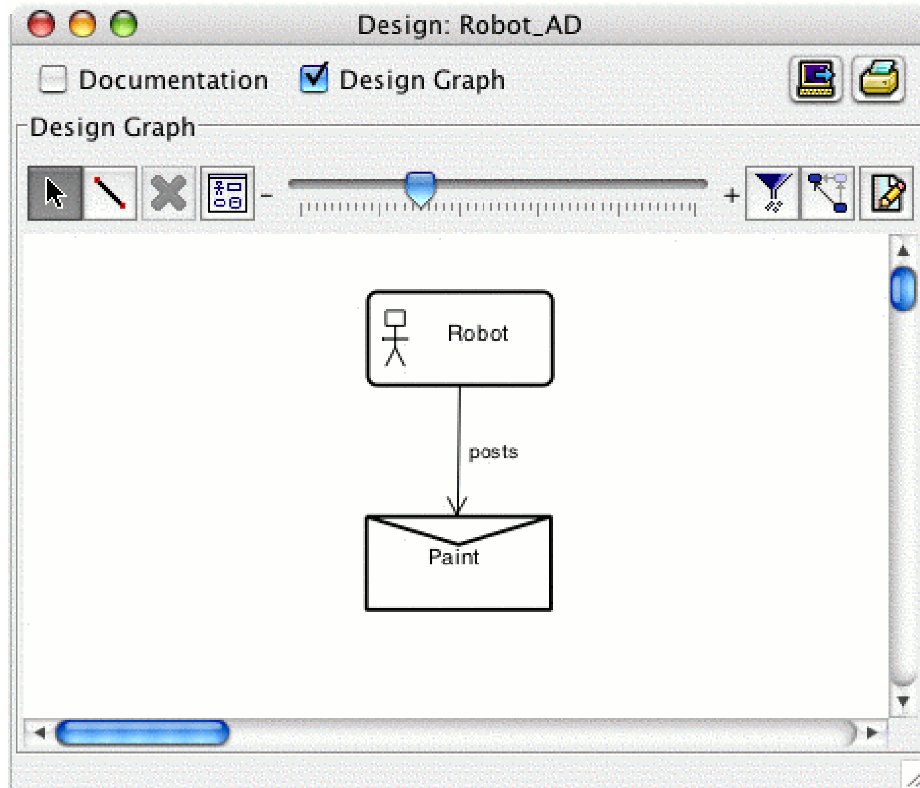

Figure 7: The `Robot_AD` design diagram with the `Robot` agent and `Paint` event

- creating a new design canvas called `Robot_cap_hier`, dragging the `Robot` agent and `Painting` capability onto the new design diagram, then creating a `has` link from the `Robot` agent to the `Painting` capability. The `Robot_cap_hier` should appear similar to the following diagram:

Copyright © 2012, Agent Oriented Software Pty. Ltd.

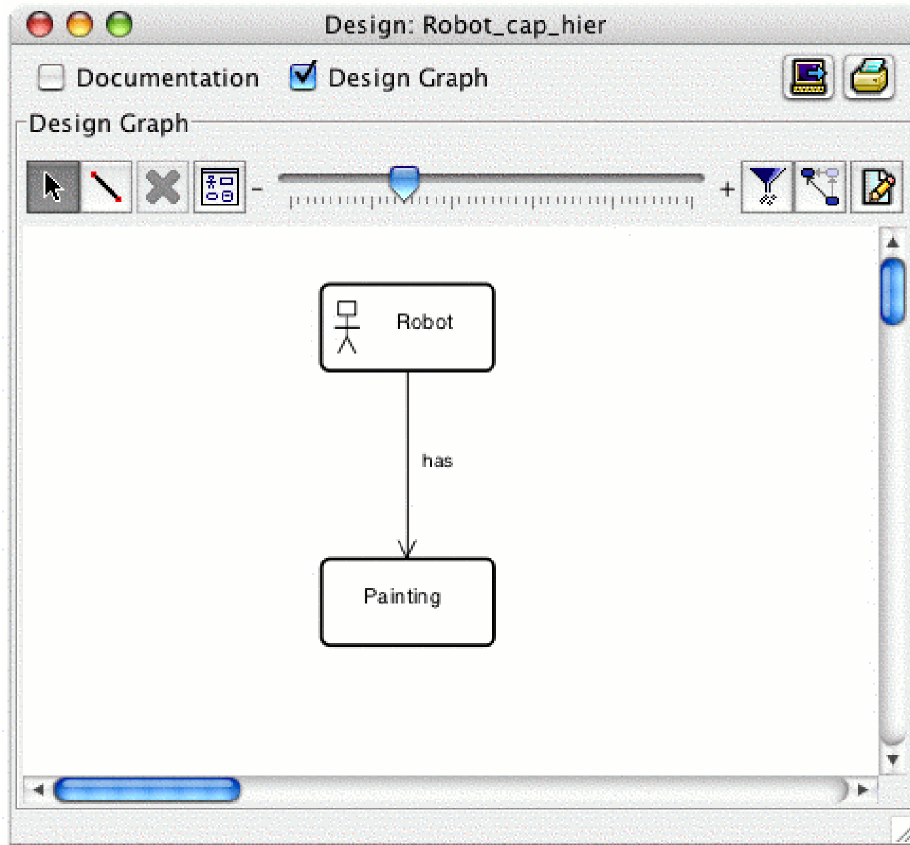

Figure 8: The `Robot_cap_hier` design diagram with the `Robot` agent and `Painting` capability

3. Edit the `Painting` capability and check that the `#uses plan` declarations are declared in the the following order:

```
#uses plan PaintSpecifiedCurrentColour;
#uses plan PaintSpecifiedNewColour;
#uses plan PaintAnyColour;
```

This is the same order as they were previously declared in the agent declaration. If editing the file as a JACK file, save and close the file before continuing.

4. Save the project.

5. Compile and run the application. Check that the output is correct. It should be similar to the following:

```
test with red
Painting part the requested colour: red
PaintSpecifiedNewColour plan failed
painting the part the current colour: red
test with no specified colour
No specified colour. Painting the part red
```


Copyright © 2012, Agent Oriented Software Pty. Ltd.

```
test with green
Painting part the requested colour (1st coat) green
PaintSpecifiedNewColour plan failed
painting the part the current colour: green
test with green again
painting the part the current colour: green
```


Copyright © 2012, Agent Oriented Software Pty. Ltd.

Exercise 7

Create a multi-agent system consisting of a robot agent and a part agent.

Introduction

In this exercise you will create a part agent that interacts with the robot agent. The part agent will have a `PaintRequesting` capability that will enable it to send painting requests to the robot agent. The part agent will post itself `PaintRequest` events (by the invocation of a `submitPaintRequest()` method in the main Java thread). The `PaintRequest` event will be handled by the part agent's `SendPaintCommand` plan. The `SendPaintCommand` plan will send a `Paint` event to the robot agent.

Note that the `Paint` event must now be a `BDIMessageEvent` (a message event is required if the event is to be sent between agents). We will also need to add

```
#set behavior Recover repost
```

to the `Paint` event. BDI events have a default behaviour that determines what happens with respect to plan reconsideration, applicable plan set recalculation and meta-level reasoning (reasoning about which plan to choose). These default behaviours can be modified by setting behaviour attributes. In this example, we set the recover attribute to `repost` which means that the event will be reposted on failure. The applicable plan set will be recomputed (with possibly different results/bindings – in this case our failed `PaintSpecifiedNewColour` plan will have changed the agent's `paintColour` member) and another applicable plan will be tried. This should become clearer when you run the example.

Instructions

7a – Organise the code into sub-folders in the JDE browser.

Before we begin adding the definitions for our `Part` agent, we will organise the code so that all the code related to the `Robot` will be in `Robot` sub-folders in the JDE browser.

It is useful to organise the project in the browser according to entities that relate to the `Robot` and entities that relate to the `Part`. This can be achieved by right-clicking on a folder/container and selecting Add Nested Container. The new nested container can be given an appropriate name (e.g. robot or part). Components can be dragged and dropped into the appropriate nested container (e.g. all the painting plans can be dragged into a nested robot container inside the Plan Types folder).

1. Organise the remainder of the project in the browser so that each folder has a nested robot folder containing the definitions related to the robot. (It is not always clear which package an event should belong to. We generally add them to the package of the agent that can handle the event.)

Copyright © 2012, Agent Oriented Software Pty. Ltd.

2. Save the project. The browser window should look similar to the following:

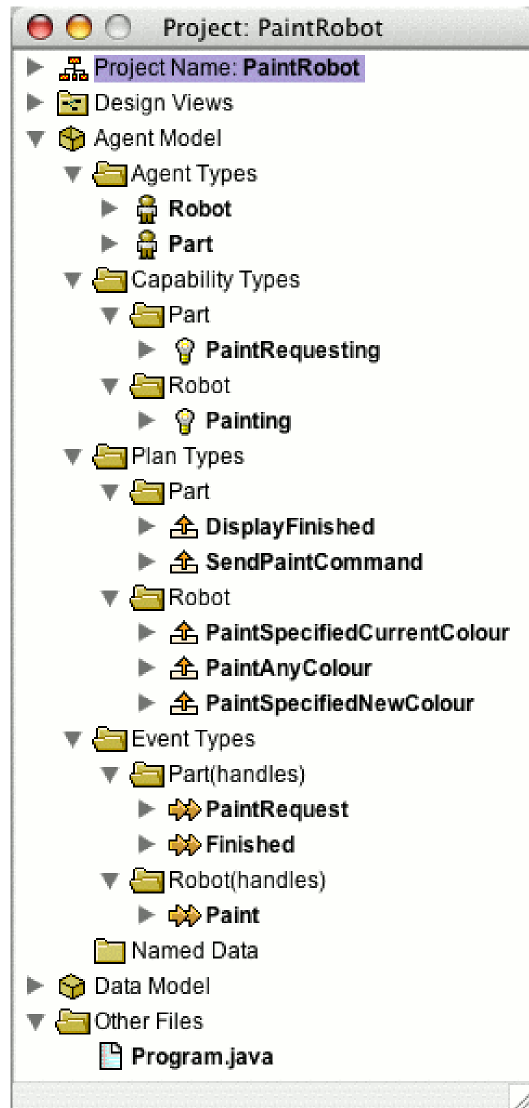

Figure 9: The browser window with the project organised into nested containers

3. Compile and run the application.

7b – Create the Part agent and incorporate it into the application

1. Create a new design diagram called `Part_AD` and
   - Drag a new agent from the design palette onto the new canvas. Name the new agent type `Part` and add it to a new `part` package.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

- Drag a new event from the design palette onto the `Part_AD` design diagram. Name the new event type `PaintRequest` and add it to the `part` package.
- On the design canvas, create a `posts` link from the `Part` agent to the `PaintRequest` event.

2. In the browser create a nested container for the Part agent in the Event Types folder. Drag the `PaintRequest` event into the Part folder. As more components (plans etc.) are added for the `Part` and `Robot`, store them in appropriate nested folders in the browser.

3. Create a new design diagram called `Part_cap_hier` and
   - Drag the `Part` agent from the browser onto the `Part_cap_hier` canvas.
   - Drag a new capability from the design palette onto the `Part-cap_hier` canvas. The new capability is called `PaintRequesting` and must be added to the `part` package.
   - Create a `has` link from the `Part` agent to the `PaintRequesting` capability.

4. Create a new diagram called `PaintRequesting_DEP` and
   - Drag a new plan from the design palette onto the new `PaintRequesting_DEP` canvas. This plan is called `SendPaintCommand` and is to be included in the `part` package.
   - Drag the `PaintRequest` event from the browser onto the `PaintRequesting_DEP` canvas.
   - Drag the `Paint` event from the browser onto the `PaintRequesting_DEP` canvas.
   - Create a `handles` link from the `PaintRequest` event to the `SendPaintCommand` plan on the `PaintRequesting_DEP` design diagram.
   - Create a `sends` link from the `SendPaintCommand` plan to the `Paint` event on the `PaintRequesting_DEP` design diagram. Note that it will be necessary to double-click on the link and change it from the default `posts` link to a `sends` link.
   - Drag the `PaintRequesting` capability from the browser onto the `PaintRequesting_DEP` design diagram. We will only have this on the diagram while we create/check the required links between the capability and its components, otherwise the design diagram will become too cluttered. While the capability is on the canvas, create the following links:
     - a `uses` link from the `PaintRequesting` capability to the `SendPaintCommand` plan.
     - a `handles` link from the `PaintRequest` event to the `PaintRequesting` capability
     - a `sends` event from the `PaintRequesting` capability to the `Paint` event.
   - Remove the `PaintRequesting` capability from the diagram. Note that the capability and the newly created links still remain in the browser. (If you were to drag the capability back onto the design diagram, you would see the links on the diagram.) Your `PaintRequesting_DEP` diagram should be similar to the following design diagram:

Copyright © 2012, Agent Oriented Software Pty. Ltd.

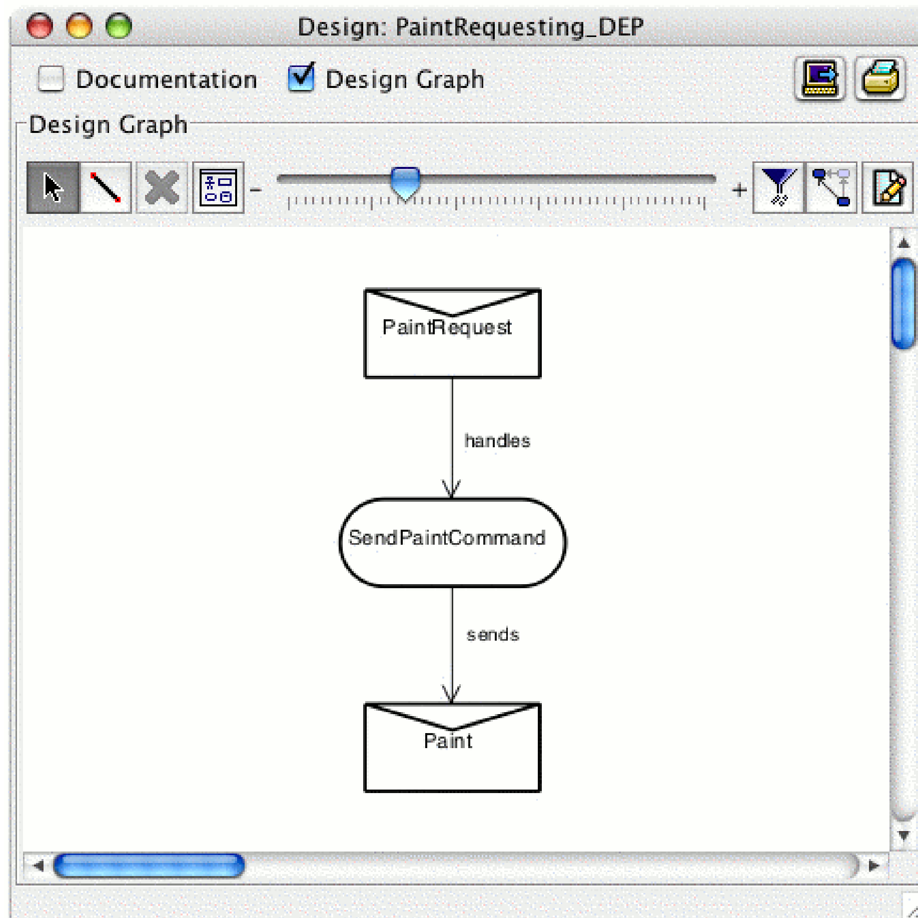

Figure 10: The `PaintReqesting_DEP` design diagram

5. Edit the `PaintRequest` event as follows. The event must

- Be a `BDIGoalEvent`;
- Have two members:

```
String robot;      // The name of robot agent to receive the
                   // paint request.
String colour;     // The colour to paint the part.
```

- Have a posting method `request(String r, String c)`. Inside the posting method, the parameter `r` should be assigned to the event data member `robot`, and the parameter `c` should be assigned to the event data member `colour`.

If editing the file as a JACK file, save and close the file before continuing.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

6. Edit the `Part` agent and add a `submitPaintRequest(String robot, String colour)` method that will post a `PaintRequest` event containing the name of the robot to send the paint request to and the colour that the part is to be painted. Use `postEventAndWait()` to post the event.

If editing the file as a JACK file, save and close the file before continuing.

7. Edit the `SendPaintCommand` plan as follows:

- Change the name of the `PaintRequest` event being handled by the plan to `preqev`.
- Change the reference to the `Paint` event type in the `sends` declaration to `pev`.
- The body of this plan must use the `@send` reasoning statement to send a `Paint` event to `preqev.robot` using the `Paint` event's `paint()` posting method. `preqev.colour` is passed as the argument to the `paint()` posting method. The send statement to be included in the `body` is therefore:

```
@send(preqev.robot, pev.paint(preqev.colour));
```

If editing the file as a JACK file, save and close the file before continuing.

8. Edit the `Paint` event to extend `BDIMessageEvent`, and set its `Recover` behaviour attribute to be `repost` (`#set behavior Recover repost`).

If editing the file as a JACK file, save and close the file before continuing.

9. Edit the `Robot` agent and remove the `paintPart()` method and the `#posts event Paint` declaration.

If editing the file as a JACK file, save and close the file before continuing.

10. Modify the `main()` method in `Program.java` so that it:

- Creates a part agent as well as a robot agent;
- Invokes the part's `submitPaintRequest()`, with the name of the robot and the required colour (instead of invoking the now defunct `paintPart()` method). The code should contain tests similar to the following:

```
System.out.println("test with red");
part1.submitPaintRequest("robbie","red");
System.out.println("test with no specified colour (null)");
part1.submitPaintRequest("robbie",null);
System.out.println("test with green");
part1.submitPaintRequest("robbie","green");
System.out.println("test with green again");
part1.submitPaintRequest("robbie","green");
```

- Includes an `import part.Part;` statement.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

11. Comment out the `System.exit(0);` statement in `Program.java`.

Save and close the file to apply the changes before continuing.

12. Save the project.

13. Compile the program.

14. Predict what you would expect to be output by the program. Run the program. Are your predictions correct?

Questions

1. What would happen if you had not added `#set behavior Recover repost` to the `Paint` event? Test your prediction by commenting out the `#set behavior` statement. Why wasn't this required when the `Paint` event was a `BDIGoalEvent`?

2. Explain the order of the output statements.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

Exercise 8

Make the part/robot messaging protocol two-way and demonstrate the use of the JACK Interaction Diagram.

Introduction

In this exercise the robot agent will send a message to the part agent when it has finished painting the part. The part agent will use a `DisplayFinished` plan to handle the `Finished` event and will print a message to indicate that it has been painted.

Instructions

1. Now that we have illustrated the reposting of events, modify the `PaintSpecifiedNewColour` plan so that it no longer fails, but instead has two painting print statements (to provide two coats of paint).

1. Open the `PaintRequesting_DEP`.
   - drag a new event onto the design diagram. This new event is to be called `Finished` and is to be in the `part` package. A `Finished` event is sent by the robot to indicate that it has finished painting the part the specified colour.
   - Drag a new plan onto the canvas. This new plan is called `DisplayFinished` and is to be in the `part` package. The `DisplayFinished` plan is to handle a `Finished` event sent to the `Part` agent by the `Robot` agent when it has finished painting the part. This will allow us to view the interaction between the agents on the interaction diagram.
   - Create a `handles` link from the `Finished` event to the `DisplayFinished` plan.
   - Drag the `PaintRequesting` capability from the browser onto the design diagram.
   - Create a `uses` link from the `PaintRequesting` capability to the `DisplayFinished` plan.
   - Create a `handles` link from the `Finished` event to the `PaintRequesting` capability.
   - Remove the `PaintRequesting` capability from the design diagram. Your `PaintRequesting_DEP` diagram should be similar to the following:

Copyright © 2012, Agent Oriented Software Pty. Ltd.

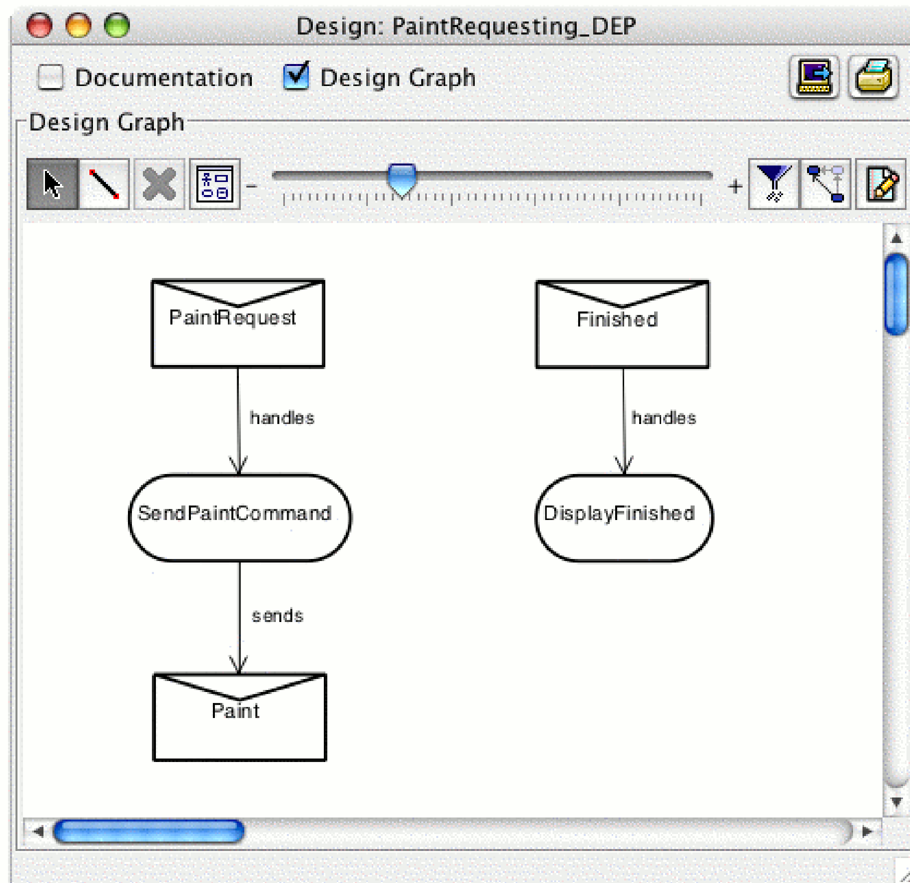

Figure 11: The `PaintReqesting_DEP` design diagram with the `DisplayFinished` plan

2. Edit the `Finished` event and complete its definition. It must:
   - extend `BDIMessageEvent`;
   - have a `String` data member called `colour`;
   - have a posting method `finished(String c)`which assigns `c` to the `colour` data member;

As this is a message event, it can display information on an interaction diagram – include the following statement in the posting method

```
message = "finished painting " + c;
```

where `c` is the colour passed to the posting method and `message` is the (inherited) `String` data member that will be displayed on the interaction diagram.

If editing the file as a JACK file, save and close the file before continuing.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

3. Edit the `DisplayFinished` plan as follows:
   - change the reference for the `Finished` event to `fev`
   - add a `#uses interface Part self;` declaration to the plan
   - add the following print statement to the `body` reasoning method of the plan:

```
System.out.println(self.name() + " has been painted " +
                   fev.colour);
```

If editing the file as a JACK file, save and close the file before continuing.

4. Open the `Painting_DEP` design diagram and
   - Drag the `Finished` event from the browser onto the canvas.
   - Create a `sends` link from each of the three painting plans to the `Finished` event. Remember that the default link type is `posts` – this will need to be changed to `sends`.
   - Drag the `Painting` capability from the browser onto the design diagram and create a `sends` link from the `Painting` capability to the `Finished` event.
   - Remove the `Painting` capability from the `Painting_DEP` design diagram. Your diagram should be similar to the following:

Copyright © 2012, Agent Oriented Software Pty. Ltd.

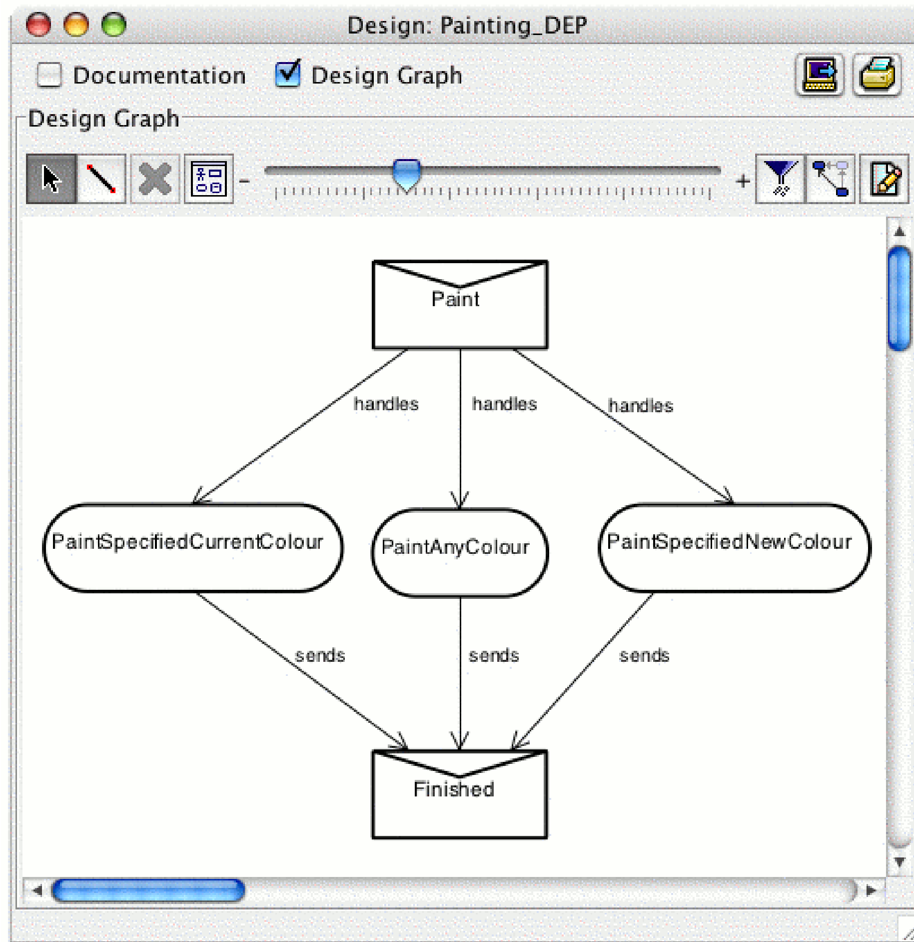

Figure 12: The `Painting_DEP` design diagram

5. Modify the `PaintAnyColour`, `PaintSpecifiedCurrentColour` and `PaintSpecifiedNewColour` plans so that when they have finished painting a part, they send the part involved a `Finished` event using:

```
@send(ev.from, fev.finished(self.paintColour));
```

where `ev` is the reference to the `Paint` event being handled by the plan, and `fev` is the reference used in the `#sends event Finished fev` declaration in the plan. `from` is an inherited `String` data member.

If editing the file as a JACK file, save and close the file before continuing.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

6. Edit the `Paint` event, so that the `message` member contains information about the paint colour. Inside the posting method assign a `String` to `message`, composed of `Paint colour` plus the colour passed to the posting method. The inherited `String` data member `message` will be displayed on the interaction diagram.

If editing the file as a JACK file, save and close the file before continuing.

7. Modify the `main()` method in `Program.java`, so that instead of one part agent there are four. Modify the code so that each `submitPaintRequest()` is associated with a different part.

Save and close the file to apply the changes before continuing.

8. Save the project.
9. Predict what you would expect to be output by the program. Compile and run your program to test your predictions. When all the parts have been painted, press the Stop button to stop the application.
10. Run the program with the interaction diagram. This is achieved by running the program with `-Djack.tracing.idisplay.type=id` set in the Java Args text box in the Compiler Utility's Run Application window.

Questions

1. Explain the order of the output from trace statements from the main program and the JACK agents.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

Exercise 9

Modify the behaviour of the robot agent so that painting takes a specific period of time to complete.

Introduction

In the previous exercise, parts were painted the requested colours. However, painting only took the amount of time required to print out a statement indicating that the robot was painting the part a particular colour. We will discover some interesting effects if we allow the plans involved to 'sleep' for a short time while the robot paints the part. To achieve this, we use the reasoning method statement `@sleep`.

The `@sleep` statement takes the following form:

```
@sleep(double timeout);
```

`timeout` represents the period of time that the agent must wait before continuing with the plan. The time-out period is specified in 'ticks' on the agent's clock. The actual time depends on the `Timer` that the agent is using. If the timer is the real-time clock (the default), then it represents a sleep period in seconds.

Note: `@sleep` only causes the current task to sleep. Any other tasks that the agent is currently executing proceed as normal.

Instructions

1. Modify the 'painting' plans so that they contain an `@sleep(5)` statement to sleep for 5 seconds after they print the message to indicate that they are painting a part.

If editing the files as JACK files, save and close them before continuing.

2. Save the project.

3. Compile and run the program with the interaction diagram.

Questions

1. How do you explain the output?

2. How can you ensure that the robot does not begin a new task to start painting another part while it is still 'busy'?

Copyright © 2012, Agent Oriented Software Pty. Ltd.

Exercise 10

Synchronise the message protocol between the part and the robot.

Introduction

In the previous example, we found that while the robot was painting a part, it was possible for it to receive requests to perform other tasks. If there is nothing in the plans to make them wait until the robot is not busy, the robot may begin a new task and change `paintColour` before it has finished painting a particular part. There are various schemes you may have thought of to deal with this. Some of them may require features in JACK that we have not yet covered, such as using a beliefset to store the robot state and waiting until the robot is no longer busy before servicing the next request.

In this exercise, we illustrate the use of `@reply` to set up a protocol between the part and the robot. The part sends the request to the robot, then waits for a reply before continuing to the end of the `SendPaintCommand` plan. The `SendPaintCommand` plan is executed in response to the `PaintRequest` event that is posted using `postEventAndWait()` inside the `submitPaintRequest()` method. As it uses `postEventAndWait()`, the method will not return until the `SendPaintCommand` plan has completed. This means that the next `submitPaintRequest()` in the main thread will not be invoked until the part involved in the previous request has received a reply to indicate that it has been painted.

Of course, if any other `Paint` events are sent to the robot from elsewhere, it is still possible to have the problems we experienced in Exercise 9. In Exercise 11, we will use an alternative solution to avoid this problem.

The `@reply` statement takes the form:

```
@reply(OriginalMessageEvent, ReplyMessageEvent)
```


Copyright © 2012, Agent Oriented Software Pty. Ltd.

The `@reply` statement is used by an agent to reply to a message that it has received from another agent. It replies to the sending agent with a message event which arrives as a data object on the reply queue of the original message. This is illustrated in the following code fragment:

```
// In the part's sending plan.

plan SendPaintCommand extends Plan
{
   #sends event Paint pev;
      :
      :

   body()
   {
      // Need to have an instance of the MyMessage event to use
      // with replied() and getReply()
      Paint q = pev.paint(...);

      //Send to robot1
      @send("robot1",q);
      @waitFor(q.replied());

      // Finished is an event defined in your system
      Finished fev = (Finished) q.getReply();

      // Do something with reply
        :
        :
   }
}

// In the robot's receiving plan
plan PaintCurrentColour extends Plan
{
   #handles event Paint pev;
   #sends event Finished rev;
             :
             :

   body()
   {
             :
             :
      @reply(pev, rev.finished(...));
             :
             :
   }
}
```

Note that the message event that is returned using `@reply` does not trigger a new task or plan.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

Instructions

1. Remove the `DisplayFinished` plan from the project by removing it from the plans folder in the browser. This is achieved by right-clicking on the plan and selecting Remove "DisplayFinished" from the pop-up menu. Note that you also need to remove the `#handles external event Finished` declaration from the `PaintRequesting` capability.

2. Modify the `SendPaintCommand` plan so that after it sends a `Paint` message to the robot it waits for a reply. When it receives the reply it uses `getReply()` to access the reply (which will be a `Finished` event). It must then print a message to indicate that it (the agent name), has been painted a particular colour. The colour is available from the reply (i.e. the `Finished` event).

If editing the file as a JACK file, save and close the file before continuing.

3. Modify the robot's paint plans, so that when they complete painting a part they use `@reply` to reply with a `Finished` event (which conveys the colour that the part was painted). Remove the `@send` statements that were used to send the `Finished` event in the earlier exercises.

If editing the file as a JACK file, save and close the file before continuing.

4. Save the project.

5. Compile and run the program with the interaction diagram.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

Exercise 11

Use a semaphore to ensure that the robot only attempts to paint one part at a time.

Introduction

In Exercise 10, the main program only sent a new paint request to the robot when the previous part had been painted. This was because the `submitPaintRequest` methods invoked from the main Java thread used the `postEventAndWait` method. The `submitPaintRequest` method did not return until the part had received a reply to indicate that the paint process had been completed. If this was guaranteed to be the only way that paint requests were sent to the robot, no conflict would arise. However, if any other paint requests were sent from the JACK thread, the robot could attempt to paint more than one part at the same time.

An alternative technique to ensure that the robot only attempts to paint one part at a time is to use a semaphore. A semaphore is a synchronisation resource which can be used to establish mutual exclusion regions of processing in JACK plans and threads. A semaphore is a binary resource that plans and threads may wait for and signal on when they have completed. Waiting entities queue on the semaphore and acquire the semaphore in FIFO order.

The semaphore has a single constructor:

```
Semaphore()
```

Methods are provided to grab and release the semaphore. `signal()` is used to release the semaphore. The semaphore is grabbed initially by the constructing thread (or plan) and must thus be released by a call to `signal()`. To grab the semaphore from within a plan, use `planWait()`. The `planWait()` method returns a special JACK type, the `Cursor`.

The cursor concept originates from relational databases, where a query can return multiple tuples in the form of a result set. Access to the elements in this set is then provided through a cursor. In JACK, these concepts have been extended to provide cursors which not only operate in the conventional manner but also operate on the temporal evolution of a query. The latter type of cursor is typically used in JACK applications to determine when a particular condition becomes true. Cursors which provide this additional capability within JACK are implemented as *triggered cursors*. Triggered cursors are not checked using a busy-wait loop – rather, they are only tested when the agent performs a modification action that impacts on the cursor. The cursor returned by the `planWait()` method is triggered when the semaphore is grabbed by the plan.

This means that a plan can use the `@waitFor` reasoning statement to wait until the plan has been able to grab the semaphore before it begins painting the part.

In this exercise we introduce a new `ProcessPaintRequest` plan. This plan can only paint a part when it has the semaphore. If it does not have the semaphore, it must wait until it gets the semaphore before it paints the part.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

Instructions

1. Create the named data of type `Semaphore` that is to be used to prevent the robot from attempting to paint more than one part at a time.

- Open the Data Model container.
- Right-click on the External Classes folder and select Add New External Class.
- In the pop-up window that appears enter `Semaphore` as the name of the external class and enter `aos.jack.util.thread` as the package.
- Click on the Add New button.
- In the Agent Model container, right-click on the Named Data folder and select Add New Named Data from the menu.
- In the pop-up window enter `mutex` as the name of the data and select `aos.jack.util.thread.Semaphore` as its type.
- Click on the Add New button.

2. Open the `Painting_DEP` design diagram and

- Remove the `sends` links from the three 'painting' plans to the `Finished` event.
- Drag a new plan from the design palette onto the `Painting_DEP` design diagram canvas. The new plan is called `ProcessPaintRequest` and is to be part of the `robot` package.
- Create a new link from the `ProcessPaintRequest` event to the `Finished` event.
- Drag a new event from the design palette onto the `Painting_DEP` design diagram canvas. The new event is to be called `StartPainting` and is to be part of the `robot` package.
- Remove the links between the `Paint` event and the three 'paint' plans.
- Create a new link from the `Paint` event to the `ProcessPaintRequest` plan.
- Create a `posts` event link from the `ProcessPaintRequest` plan to the `StartPainting` event.
- Create `handles` links from the `StartPainting` event to each of the three 'painting' plans.
- Drag the `mutex` named data onto the canvas.
- Create a link from the `ProcessPaintRequest` plan to the `mutex` named data.
- Drag the `Painting` capability onto the `Painting_DEP` design diagram.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

- Create the following links:
  - A uses link from the Painting capability to the ProcessPaintRequest plan.
  - A handles link from the StartPainting event to the Painting capability.
  - A posts link from the Painting capability to the StartPainting event.
  - A #private link from the Painting capability to the mutex data.
- Remove the Painting capability from the Painting_DEP design diagram. Your Painting_DEP should be similar to the following diagram:

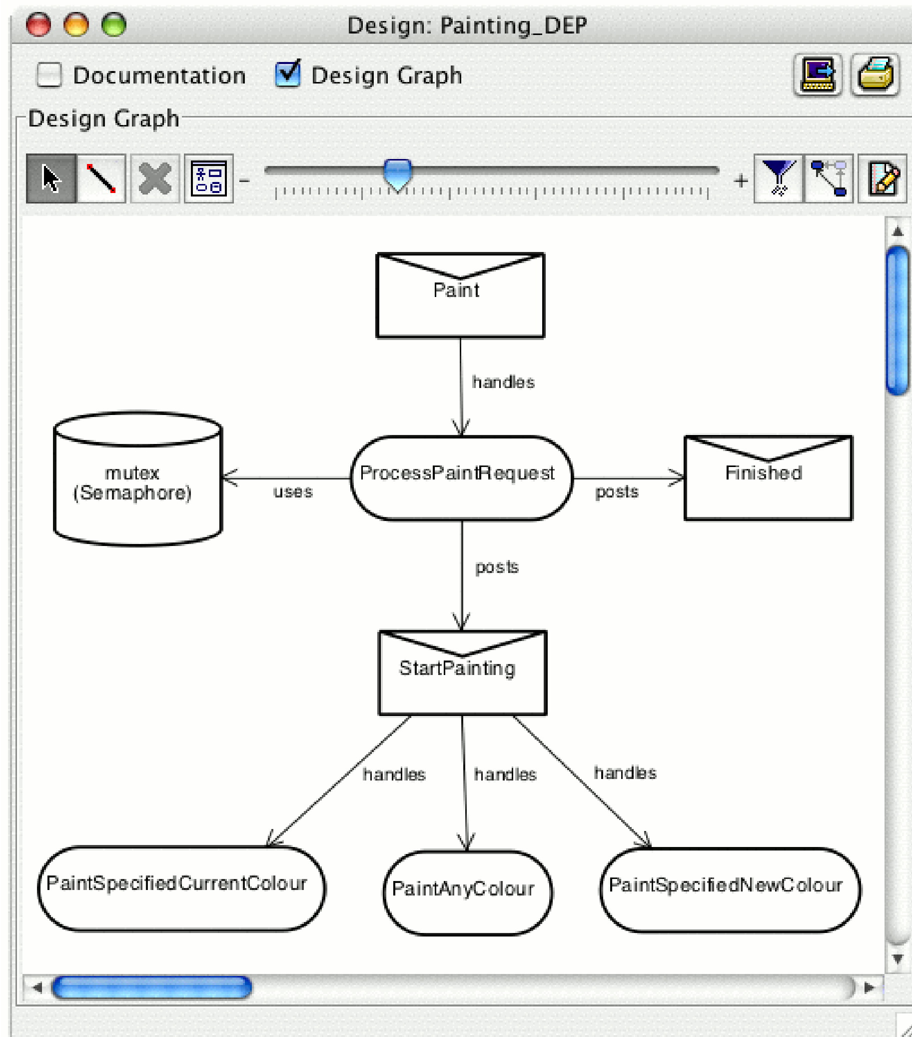

Figure 13: The Painting_DEP design diagram with the ProcessPaintRequest plan, StartPainting event and mutex named data

3. Edit the StartPainting event so that it:

Copyright © 2012, Agent Oriented Software Pty. Ltd.

- Extends `BDIGoalEvent`.
- Has a `String` member `colour` and a `String` member `from`.
- Has a posting method called `startPainting`:

```
startPainting(String c, String f)
{
   colour = c;
   from = f;
}
```

If editing the file as a JACK file, save and close the file before continuing.

4. As the semaphore is grabbed initially by the constructing thread, it must also be released by a call to `signal()`. The `#private data Semaphore mutex();` declaration constructs the semaphore in the capability. `Painting` capability. To release the semaphore after it has been constructed, edit the capability and override its `autorun` method as follows:

```
protected void autorun()
{
   mutex.signal();
}
```

5. Edit the `ProcessPaintRequest` plan as follows:

- Edit the declarations at the beginning of the plan to use the following event references:

```
#handles event Paint pev;
#posts event StartPainting spev;
#sends event Finished fev;
```

- Edit the `body` of the plan. This plan waits until it owns the semaphore before it subtasks the appropriate 'painting' plan to paint the part. When the 'painting' subtask is complete, the `ProcessPaintRequest` plan must send a `Finished` event back to the part. In addition, when the subtask is complete, this plan must release the semaphore. The `body` should therefore be as follows:

```
@waitFor(mutex.planWait()); // Wait for semaphore (mutex).

try {
   @subtask(spev.startPainting(pev.colour, pev.from));
   @send(pev.from, fev.finished(pev.colour));
}
finally {
   mutex.signal(); // Release semaphore (mutex).
}
```

If editing the file as a JACK file, save and close the file before continuing.

6. Use the browser to remove the `@reply` statements from the three 'painting' plans.

7. Remove the `@waitFor` reply and associated statements from the `Part` agent's `SendPaintCommand` plan.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

8. Open the `PaintRequesting_DEP` design diagram and

   – Drag a new plan from the design palette onto the `PaintRequesting_DEP` design diagram canvas. The new plan is to be called `DisplayFinished` and is to be part of the `part` package.

   – Create a `handles` link from the `Finished` event to the `DisplayFinished` plan.

   – Drag the `PaintRequesting` capability onto the `PaintRequesting_DEP` design diagram canvas.

   – Create the following links:

     - A `uses` link from the `PaintRequesting` capability to the `DisplayFinished` plan.
     - A `handles` link from the `Finished` event to the `PaintRequesting` capability.

   – Remove the `PaintRequesting` capability from the `PaintRequesting_DEP` design diagram. Your `PaintRequesting_DEP` diagram should be similar to the following:

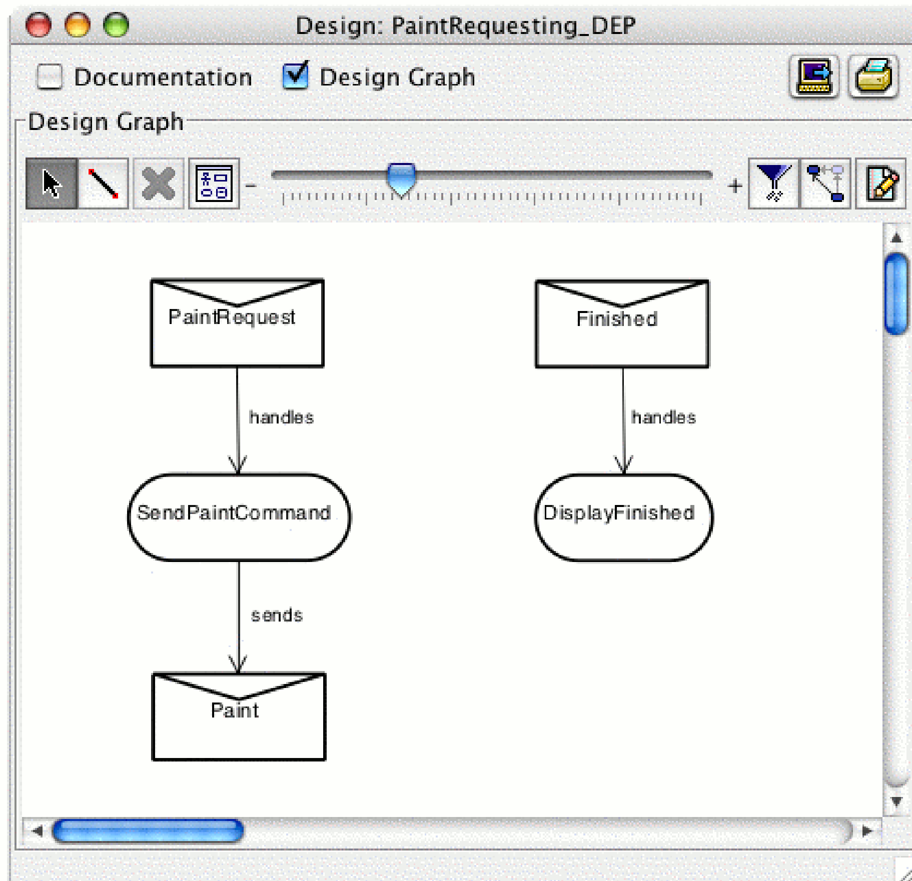

Figure 14: The `PaintReqesting_DEP` design diagram with the `DisplayFinished` plan and `Finished` event

Copyright © 2012, Agent Oriented Software Pty. Ltd.

9. Edit the `DisplayFinished` plan and

   – modify the `#handles` declaration to use `fev` as the reference to the `Finished` event handled by the plan.

   – Add a `#uses interface Part self;` declaration.

   – Add the details to the `body` of the plan. All that this plan needs to do is to display a message to indicate that the part has been painted a particular colour. It should contain a print statement similar to the following:

   ```
   System.out.println(self.name()+" has been painted "
                       +fev.colour);
   ```

   where `fev` is the reference to the `Finished` event handled by the plan.

If editing the file as a JACK file, save and close the file before continuing.

10. Save the project.

11. Compile and run the project.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

Practical 2 JACK Beliefset Relations

Introductory notes

JACK beliefs

JACK beliefset relations are intended to be used for the maintenance of agent beliefs, which are central to the notion of BDI. Normal Java data structures can also be used for this purpose, but JACK beliefset relations have been designed specifically to work within the agent-oriented paradigm, and provide facilities not available with other data storage techniques. These additional facilities include:

- Automatic maintenance of logical consistency and key constraints.
- A choice of Open World semantics (where a belief is either true, false or unknown) or Closed World semantics (where a belief is either true or false).
- The ability to post events automatically when beliefs change in the beliefset.
- Support for beliefset cursors and logical members, which enables beliefset access to be implemented through unification.

Note that JACK only provides the infrastructure for adding, removing and retrieving beliefs. If other capabilities are required (such as belief propagation), then they must be implemented by the user. Also note that beliefs are associated with one agent only – the notion of a collective belief structure which is maintained by more than one agent is not supported. JACK allows an agent's beliefs to be made available to other agents but with major restrictions regarding population and access.

In JACK, beliefs are modelled as tuples. Every tuple must have a unique key which is composed of zero or more fields. In addition to the key, the tuple contains zero or more data fields. If the key contains zero fields, only one tuple is allowed in the beliefset. It is permissible for the key to be composed of all fields in the tuple.

A JACK beliefset is populated using the `BeliefSet add()` method. Retraction and retrieval of beliefs also follow the relational model in that both functions support the notion of tuple specification. A tuple specification consists of components for each tuple field where each component can either be a value or a wildcard symbol (e.g. *) which indicates that any value for that field is acceptable.

In JACK, tuple specification is implemented as an argument list consisting of JACK expressions (corresponding to values) and unbound logical members (corresponding to wildcards). Logical members follow the semantic behaviour of variables from logic programming languages such as Prolog, where variable binding occurs through the process of unification. Unification can potentially provide multiple bindings for the unbound logical members – JACK provides access to these bindings (if required) through a beliefset cursor.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

This access can be transparent to the user (e.g. when a beliefset query appears in a composite logical expression) or under user control. In the latter case, the cursor method `next()` is available to provide access to the next binding.

Once a logical member has been bound, its value cannot be changed and an accessor function (e.g. `getValue()`) is required to access the value of the bound logical member.

Belief retraction is achieved in JACK through the `remove()` and `removeAll()` methods. Note that `remove()` is a `BeliefSet` method (and takes a tuple specification as its argument list) whereas `removeAll()` is a `BeliefSetCursor` method and takes no arguments.

Belief retrieval is achieved in JACK through user defined query methods which take a tuple specification as their argument list. Overloading of query methods is permitted. A query method with an argument list consisting solely of (non-logical) JACK expressions is valid – it is used to determine whether or not a particular belief is held. JACK also supports (through the view construct) the retrieval of beliefs which are constructed from multiple beliefsets.

Once asserted, a JACK belief cannot be modified directly. The only way to change a belief is to:

- add a revised belief; or
- remove the belief.

JACK supports the posting of events when an agent's beliefs change (e.g. through an `add()` or `remove()`). The following beliefset callbacks are available for this purpose – `addfact()`, `newfact()`, `delfact()`, `endfact()` and `modfact()`. If this capability is required in a particular application, the developer must provide implementations for the callbacks that are actually used.

If you wish to populate a beliefset from a data file, then you can read records from a data file and explicitly add the records to the beliefset using `add()`. Typically, you would perform this activity from within the beliefset constructor, with the filename passed as an argument.

Alternatively, you can initialise tuple objects directly using the JACOB Object Modelling Language, which is described in the *JACK™ Intelligent Agents JACOB Manual*. Beliefsets are provided with a `read()` method which takes a JACOB file as its argument and populates the beliefset according to the contents of the file. JACOB can be used for initialising any JACK objects, not just beliefsets and is also used to exchange objects between processes. In this latter capacity, it provides a lightweight alternative to Java serialization and CORBA.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

JACK beliefset definition

The general form for a Closed World JACK beliefset definition is:

```
beliefset RelationName extends ClosedWorld
{
  // JAL declarations.
  // The following declarations can be used in a
  // JAL beliefset relation:
  //
  // Zero or more key field declarations.
  #key field FieldType FieldName;
  // Zero or more value field declarations.
  #value field FieldType FieldName;

  // Declaration of any events that are posted when changes are
  // made to the tuples. This only happens when a beliefset callback
  // is defined.
  #posts event EventType ref;

  // Declaration of queries that the agent can perform on a relation.
  // They can include:
  #indexed query queryName( parameter list);
  #linear query queryName(parameter list);
  #complex query queryName(parameter list) {method body}
  #function query ReturnType queryName(parameter list) { method body}
  // Note that only the prototype is declared for indexed and linear
  // queries - the actual query class (which will be a beliefset
  // cursor) is generated automatically by the JACK compiler.

  // Beliefset callbacks (if required). The prototypes are:
  public void addfact(Tuple t,BeliefState is)
  // Called when a belief is added.
  public void newfact(Tuple t,BeliefState is,BeliefState was)
  // Called when a new belief is added.
  public void delfact(Tuple t,BeliefState was)
  // Called when a belief is removed.
  public void endfact(Tuple t,BeliefState was,BeliefState is)
  // Called when a belief is removed.
  public void modfact(Tuple t,BeliefState is,
                      Tuple knocked,Tuple negated)
  // Called when a belief is modified.
  public void moddb()
  // Catch-all.
}
```

The open world's beliefset relation definition takes the same form as that shown for the closed world except that in the first line ClosedWorld is replaced with OpenWorld.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

An example

The following JACK beliefset could be used to maintain beliefs regarding politicians.

```
beliefset Politician extends OpenWorld
{
  #key field String name;
  #value field String party;
  #value field String portfolio;

  #indexed query get(
        String name,String party,logical String portfolio);
  #indexed query get(
        logical String name,String party,String portfolio);
  #indexed query get(
        String name,logical String party,logical String portfolio);
  #indexed query get(
        logical String name,String party,logical String portfolio);

  #posts event Elected e;

  // Callback is invoked when a new belief is added.
  public void newfact(Tuple t, BeliefState is, BeliefState was)
  {
     // The Politician__Tuple class is generated by JACK.
     // NB: Politician__Tuple has two underscores.
     Politician__Tuple pt = (Politician__Tuple) t;

     // Only post an event if the politician added is the prime
     // minister.
     if(pt.portfolio.equals("Prime Minister"))
        postEvent(e.newPrimeMinister(pt.name,pt.party,pt.portfolio));
  }
}
```

An agent can then have read/write access to a beliefset of this type by including a declaration like the following in the agent definition:

```
 #private data Politician mps();
```

The agent might populate the beliefset through a plan called `NewMember`:

```
plan NewMember extends Plan
{
  #handles event elected e;
  #modifies data Politician mps;

  body()
  {
      mps.add(e.name,e.party,e.portfolio);
  }
}
```


Copyright © 2012, Agent Oriented Software Pty. Ltd.

Other plans might then access the beliefset in various ways.

```
// Example 1.
// Is Humphrey Bear, a member of the Honey Party, Prime Minister?
// Note: the following code will only work as intended if there can
// only be one Prime Minister. Why?
    logical String name;
    mps.get(name,"Honey Party","Prime Minister");
    if (name.getValue().equals("Humphrey Bear")
    {
      // Do something
    }

// Example 2.
// Are there any elected members of the Honey Party whose first
// name is Rupert?
// What happens in this example if the first match found is not
// Rupert?
// Refer to the section on Composite Logical Expressions in the
// Plans chapter in Agent Manual for a more detailed discussion.
    logical String name;
    logical String portfolio;
    if (mps.get(name,"Honey Party",portfolio) &&
             (name.getValue().indexOf("Rupert") == 0))
    {
      // Do something.
    }

// Example 3.
// Have a picnic for all elected members of the Honey Party when
// Paddington Bear gets elected.

  // Wait until Paddington Bear is elected.
  logical String portfolio;
  #posts event SendInvitations sie;
  @waitFor(mps.get("Paddington Bear","Honey Party",portfolio));
  @post(sie.invite("Honey Party"));

//The sending of invitations is achieved with the following plan
//and event.
plan PrintInvitations extends Plan
{
  #handles event SendInvitations si;
  #uses data Politician mps;
  logical String name;
  logical String portfolio;

  context()
  {
     mps.get(name, si.politicalParty, portfolio);
  }

  body()
  {
    System.out.println("send invitation to "+name.getValue());
  }
}
```


Copyright © 2012, Agent Oriented Software Pty. Ltd.

```
event SendInvitations extends InferenceGoalEvent
{
  String politicalParty;

  #posted as invite(String p)
  {
    politicalParty = p;
  }
}

// Example 4.
// Sack Little Bear.
  mps.remove("Little Bear", "Honey Party", "Treasurer");

// Example 5.
// Sack the entire Honey Party.
  logical String name;
  logical String portfolio;
  mps.get(name,"Honey Party",portfolio).removeAll();
```

Note: You may now complete Practical 2 exercises 1 to 5.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

Exercise 1

Set up a Bill of Materials (BOM) beliefset.

Introduction

In this exercise you will create a simplified BOM beliefset to be used within a `Planner` agent. A BOM is a data structure used in manufacturing to describe the component/subcomponent structure of part assemblies. In this tutorial, the BOM will only capture the component/subcomponent structure: in practice, it will contain much more information, such as the number of a particular component that is required, whether the component is made internally or is outsourced etc.

You will also create a main program to read BOM data provided by the user. This information will be stored in a JACK beliefset which is private to the `Planner` agent. An alternative mechanism for initialising the beliefset is to use a JACOB file. This alternative is used in Exercise 5.

Instructions

1. Create a directory called `practical2/ex1`. In this directory, start the JDE and open a new project called BOM.

2. Create a beliefset type called `BOM`:

- Open the Data Model container in the browser.
- Right-click on the Beliefset Types folder and select Add New Beliefset Type from the pop-up menu. Call the beliefset `BOM` and add it to the `bom` package. It is to have two key fields of type `String`. One field is used to store a component type, and the other field is used to store a subcomponent type. This means that in this beliefset there will be *n* entries per component, where *n* is the number of subcomponent types that are used to build that component. Note that we distinguish between subcomponent types and subcomponents. For example, a table may have two subcomponent types (top and leg) but five subcomponents (one top and four legs).
- Use the Edit as JACK File option to add the two key fields and a `getSubcomponent` query to the beliefset. In Exercise 2 the `getSubcomponent` query will be used to find a subcomponent of a given component. This means that the subcomponent field is to be an output field in the query. Although this query is not required until Exercise 2, it is added now because a JACK beliefset must have at least one query. Your `BOM` beliefset should be similar to the following:

Copyright © 2012, Agent Oriented Software Pty. Ltd.

```
package bom;

public beliefset BOM extends OpenWorld {
    #key field String component;
    #key field String subcomponent;
    #indexed query getSubcomponent(String component,
                            logical String subcomponent);
}
```

3. We will now use the design tool to define an agent, `Planner`, that will store component/subcomponent tuples in a private beliefset of type `BOM`:

– Create a new design diagram called `Planner_AD`.

– Drag a new agent onto the `Planner_AD` design canvas. It is to be called `Planner` and it is to be in the `bom` package.

– Drag the Named Data icon from the design palette onto the design canvas. The named data is to be called `bom` and is to be of type `bom.BOM`.

– Create a `private` link from the `Planner` agent to the `bom` named data on the design canvas.

4. The beliefset will be populated at agent construction time. Consequently the agent's constructor will be passed two strings – the agent's name and the name of a file containing component-subcomponent details. The constructor must read each component-subcomponent relation from the file, and add the information to the agent's `BOM` beliefset. The code to achieve this can be added to the agent by using the browser to edit the agent as a JACK file. Sample code for the constructor follows:

```
public Planner(String name, String  filename)
{
   super(name);

   StringTokenizer tokens;
   String record;
   String t1,t2;
   BufferedReader datafile = null;

   try {
      datafile = new BufferedReader(new FileReader(filename));
   }
   catch (FileNotFoundException e) {
      System.err.println("unable to open file "+e.toString());
      System.exit(1);
   }

   try {
      while ((record = datafile.readLine())!=null) {
         tokens = new StringTokenizer(record);
         t1 = tokens.nextToken();
         t2 = tokens.nextToken();
         bom.add(t1,t2);
      }
   }
```


Copyright © 2012, Agent Oriented Software Pty. Ltd.

```
      catch (Exception e) {
         System.err.println("error reading bom data into beliefset");
         System.exit(1);
      }
   }
```

Make sure that the `Planner` agent has the following import statements:

```
import java.io.*;
import java.util.*;
```

5. Add a new file called `Program.java` to the Other Files folder in the browser. It will create a `Planner` agent, perhaps as follows:

```
import java.io.*;
import bom.Planner;

public class Program {
   public static void main( String args[] )
   {
      Planner planner = new Planner("planner1","bom.dat");
      System.exit(0);
   }
}
```

If you wish to run the project from within the JDE make sure that you use the full pathname for bom.dat.

6. Create the file `bom.dat` and populate it with component/subcomponent records. It could contain:

```
chair back
chair seat
chair leg
chair arm
table top
table leg
cupboard door
cupboard drawer
cupboard leg
cupboard cabinet
```

7. Compile and run the program. No output is generated by the program at this stage, but it provides the basis for the remaining exercises.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

Exercise 2

Use the indexed query to find a component's subcomponents.

Introduction

This step involves the use of indexed queries and a plan with a `context()` method to print out one of the subcomponents of a component. Initially we use a `BDIGoalEvent` so that only the first successful applicable plan is executed. This means that if we post only one event (e.g. find a subcomponent of component X), the first plan to have a context that finds a binding for a subcomponent of X will be executed. In that case we will only find (at most) one subcomponent.

Instructions

1. Open the `Planner_AD` design diagram and
   - Drag a new event from the design palette onto the design canvas. This event is to be called `FindSubcomponent` and is to be added to the `bom` package.
   - Create the following links with the `Planner` agent:
     - A `handles` link from the `FindSubcomponent` event;
     - A `posts` link to the `FindSubcomponent` event;
2. In the `Planner_AD` design diagram:
   - Add a new plan called `FindSubcomponentPlan` that is in the `bom` package.
   - Create a `handles` link between the `FindSubcomponent` event and the `FindSubcomponentPlan` plan.
   - Create a `uses` link from the plan to the `bom` named data.
   - Create a `uses` link from the `Planner` agent to the `FindSubcomponentPlan` plan.
3. Edit the `FindSubcomponent` event so that it:
   - extends `BDIGoalEvent`;
   - has a `String` data member `component`; and
   - includes a posting method `findSubcomponent(String component)` that assigns data to the `component` data member.
4. Edit the `FindSubcomponentPlan` plan so that it
   - Has a logical variable `logical String $subcomp`;

Copyright © 2012, Agent Oriented Software Pty. Ltd.

- Includes a `context` method that uses the BOM beliefset's `getSubcomponent()` query to find a subcomponent for the component passed in with the event. `$subcomp` is passed into the query. (Note: the plan will only be applicable if the `getSubcomponent()` query can find an entry for the `component` in the BOM beliefset);
- Includes a print statement similar to the following in the body of the plan:

```
System.out.println($subcomp.getValue()+" is a subcomponent of "
                                +ev.component);
```

5. Modify the `Planner` agent so that it contains a `findSubcomponent(String component)` method which will post a `FindSubcomponent` event using `postEventAndWait()`.

6. Modify the main program so that it makes several invocations of the `Planner` agent's `findSubcomponent()` method with the name of a component that is in the agent's BOM beliefset. Ensure that for at least one of the invocations the component is composed of several subcomponent types.

7. Compile and run the program.

Questions

1. Why aren't all the component's subcomponents printed out?

Copyright © 2012, Agent Oriented Software Pty. Ltd.

Exercise 3

Use the `InferenceGoalEvent` class instead of `BDIGoalEvent`.

Introduction

In this example, you will use an `InferenceGoalEvent` instead of the `BDIGoalEvent`. This illustrates two concepts. The first is the difference between these two types of event (i.e. the `InferenceGoalEvent` will cause all applicable plans to be executed). Secondly, it shows that there is a separate plan instance generated in the applicable set for each of the possible bindings in the context method. This means that in this example we will see all the subcomponents being printed. A separate plan instance will be responsible for printing the message about each subcomponent.

Instructions

1. Modify the `FindSubcomponent` event so that it is an `InferenceGoalEvent` and not a `BDIGoalEvent`.

2. Compile and test the program. The tests should involve at least one component which is composed of four or more subcomponents.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

Exercise 4

Use a beliefset callback method.

Introduction

In this exercise you add an order beliefset to your `Planner` agent and use a beliefset callback method to post a `FindSubcomponent` event to the agent. This will result in a list of subcomponents being printed for the required new component.

Instructions

1. Use the JDE to create a new beliefset type called `Orders` that belongs to the `bom` package. It will have one key field (`String orderId`) and three non-key fields (`String component`, `int numberRequired`, `String date`). Add at least one query to the beliefset.

2. Open the `Planner_AD` design diagram and

   – Drag a new named data component from the design palette onto the design diagram canvas. Call the named data `orders` and set its type to be `bom.Orders`.

   – Create a `private` link from the `Planner` agent to the `orders` named data.

3. Modify the `Planner` agent so that it contains a new method `addOrder(String orderId, String component, int numberRequired, String dueDate)` that will add a new order into the `orders` beliefset. Note that the `add()` method may throw a `BeliefSetException`, so `add` must be invoked from inside a `try/catch` block. You will also need to import `aos.jack.jak.beliefset.BeliefSetException`. Your `addOrder()` method should be similar to the following:

```
public void addOrder(String id, String comp, int n, String dd)
{
      try {
         orders.add(id,comp,n,dd);
      }
      catch (BeliefSetException e) {
         System.err.println("error entering data in orders db");
         System.exit(1);
      }
}
```

4. Edit the `Orders` beliefset type and

   – add a `#posts event FindSubcomponent ev;` declaration;

   – Add an indexed query that accepts the order ID (`String`), component type (`logical String`), number of components required (`logical int`) and the order date (`logical String`).

Copyright © 2012, Agent Oriented Software Pty. Ltd.

- Add a `newfact` callback to the `orders` beliefset. Each time a new fact is added to the beliefset, the callback will print a trace message to indicate that a new order has been added to the beliefset and then post a `FindSubcomponent` event. An example of a beliefset that includes a `newfact` callback can be found in the Politician example in the introductory notes.

5. Modify the main program so that it:

   - includes the following import statement:

```
import java.util.*;
```

   - no longer invokes the `Planner` agent's `findSubcomponent` method;
   - no longer exits; and
   - reads in the order data from a datafile and invokes the `Planner` agent's `addOrder` method to add the details to the beliefset. Sample code for this follows:

```
StringTokenizer tokens;
String record;
String t1,t2,t3,t4;
BufferedReader datafile = null;
Planner planner = new Planner("planner1","bom.dat");

try {
   datafile = new BufferedReader(new FileReader("orders.dat"));
}
catch (FileNotFoundException e) {
   System.err.println("unable to open file "+e.toString());
   System.exit(1);
}

try {
   while ( (record = datafile.readLine()) != null ) {
      tokens = new StringTokenizer(record);
      t1 = tokens.nextToken();
      t2 = tokens.nextToken();
      t3 = tokens.nextToken();
      t4 = tokens.nextToken();
      planner.addOrder(t1,t2,Integer.parseInt(t3),t4);
   }
}
catch (Exception e) {
   System.err.println("error reading orders data ");
   System.exit(1);
}
```

6. Create a data file that contains several orders. At least one order should be for a component that has four or more subcomponents.

7. Compile and run the program.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

Exercise 5

Use a JACOB initialisation file to initialise the `BOM` beliefset.

Introduction

In this exercise, you will modify the `Planner` agent's constructor so that it no longer reads the data from the text file used in the earlier exercises. Instead, you will create a file in JACOB ASCII format that is read in either when the beliefset is constructed, or by using the beliefset `read` method in the agent's constructor.

Instructions

1. Edit the `Planner.agent` and remove the code to read the BOM data from the filename passed into the constructor.

2. Modify the `#private data BOM bom()` declaration so that the filename is now passed to the `BOM` constructor (`#private data BOM bom("bom.dat")`).

3. Edit the file `bom.dat` so that it contains the data in JACOB ASCII format. An example data file is shown below.

```
<TupleTable
   :tuples(
      <BOM__Tuple
         :component "table"
         :subcomponent "leg"
      >
      <BOM__Tuple
         :component "table"
         :subcomponent "table_top"
      >
   )
>
```

4. Compile and run the program.

5. It is also possible to explicitly invoke the beliefset's `read` method to read data in JACOB ASCII format. This means that the filename can then be passed as a parameter to the agent's constructor.

   – If necessary, modify the `Planner.agent` so that the filename is passed in to its constructor.

   – Add the line

```
bom.read(filename);
```

   in the `Planner` agent's constructor.

   – Remove the filename from the `#private data BOM bom("bom.dat")` declaration.

   – Compile and run the program.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

Practical 1 Solutions

Program solutions

The solutions to the programming exercises can be found in the `practicals/jack_jde/solutions/practical1` subdirectory. There is a separate directory for each of the exercises.

Answers to questions

Exercise 2

1. The `paintPart` method would return immediately. It would not wait for the event to be processed. As the program contains a `System.exit(0)`, it would exit before any painting was observed.
2. The agent would be blocked. JACK will produce an error message and prevent this.

Exercise 3

1. Prominence, i.e. the plan declared first.

Exercise 5

1. `PaintSpecifiedCurrentColour`.

Exercise 7

1. The plan fails after the first coat of paint. The second coat would not be applied because the event is not reposted. The `#set behavior Recover repost` is the default behaviour for a `BDIGoalEvent`.
2. The main thing to notice is that print statements in the main thread that follow `submitPaintRequests` can be executed before the events sent by `submitPaintRequests` are processed. The JACK processing is in another thread.

Exercise 8

1. Observation of behaviour that is similar to that explained in exercise 7, with additional trace information caused by the `Finished` event. In addition to the trace statements, messages should have been observed on the interaction diagram.

Exercise 9

1. It takes so long to paint a part that the next request is sent (and received) before the painting process is finished. This means you do not always achieve the desired/expected output.
2. The solution could make use of a semaphore. This is explored further in exercise 11.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

Practical 2 Solutions

Answers to questions in introductory notes

There are some questions in the comments of the examples.

1. Example 1.

```
// Example: Is Humphrey Bear, a member of the Honey Party,
// Prime Minister?

// Question:  The following code will only work as intended if there
// can only be one Prime Minister. Why?

// Answer: If there is more than one prime Minister,
// Humphrey Bear will not necessarily be the first match that will
// bind to name.  The fact that name does not equal Humphrey Bear
// does not tell us whether or not there is another entry in the
// beliefset with Humphrey Bear as the Prime Minister.

    logical String name;
    mps.get(name,"Honey Party","Prime Minister");
    if (name.getValue().equals("Humphrey Bear")
    {
      // Do something.
    }
```

2. Example 2.

```
// Example: Are there any elected members of the Honey Party
// whose first name is Rupert?

// Question: What happens in this example if the first match found is
// not Rupert?

// Answer: The agent does not fail the logical expression at that
// point. It tests it again with a new binding for name, until
// either it finds a binding with Rupert or it has checked all the
// beliefset relations and concluded that Rupert is not in the
// Honey Party.

// Refer to the section on Composite Logical Expressions in the
// Plans chapter in Agent Manual for a more detailed discussion.

    logical String name;
    logical String portfolio;
    if (mps.get(name,"Honey Party",portfolio) &&
             ((name.getValue().indexOf("Rupert")==0 ))
    {
        // Do something.
    }
```

Program solutions

The solutions to the programming exercises can be found in the `practicals/jack_jde/solutions/practical2` subdirectory. There is a separate directory for each of the exercises.

Copyright © 2012, Agent Oriented Software Pty. Ltd.

Answers to questions

Exercise 2

1. The event triggers the first applicable plan which successfully prints out the message. As it was successful, there is no reposting or triggering of alternative plans with different bindings.

Copyright © 2012, Agent Oriented Software Pty. Ltd.