Ponder2Project

Ponder2

Contents

Ponder2 combines a general-purpose, distributed object management system with a Domain Service, Obligation Policy Interpreter, Command Interpreter and Authorisation Enforcement. The Domain Service provides an hierarchical structure for managing objects. The Obligation Policy Interpreter handles Event, Condition, Action rules (ECA). The Command Interpreter accepts a set of commands, compiled from a high-level language called PonderTalk, via a number of communications interfaces which may perform invocations on a ManagedObject registered in the Domain Service. The Authorisation Enforcement caters for both positive and negative authorisation policies, provides the ability to specify fine grained authorisations for every object and implements domain nesting algorithms for conflict resolution.

Ponder is the name of a policy specification language developed at Imperial College over a number of years. A set of tools and services were developed for the specification, analysis and enforcement of these policies. Thus, the name *Ponder* became associated not only with the language but with the entire toolkit. *Ponder2* is a significant re-design and re-implementation of Ponder. Although some of the underlying concepts bear similarity to the basic constructs of Ponder the entire framework has been re-done. In contrast to the previous version, which was designed for general network and systems management, Ponder2 has been designed as an entirely extensible framework that can be used at different levels of scale from small, embedded devices to complex services and Virtual Organisations.

Ponder2 has been realised with financial support from several sponsors.

Documentation

The following documents will assist you in downloading, installing and running Ponder2.

- Ponder2Overview A brief, high level, overview of Ponder2
- Installation and Running Guide How to get Ponder2 and start it
- Using Ponder2 How to write and send commands to Ponder2
- Using Ponder2 with Eclipse How to configure Eclipse for use with Ponder2
- PonderTalk The complete language reference
- Writing Your Own Managed Objects Guide to using Java with Ponder2
- Ponder2 Policies Descriptions of the policy types supported by Ponder2
- Ponder2 Events How to define and create events for Policies
- Ponder2 Authorisation Description with examples of authorisation constraints
- The Ponder2 Shell How to interact with Ponder2
- Ponder2 Communications How Ponder2 uses objects from another Ponder2 instantiation
- Ponder2 External Communications How to interface Ponder2 to other applications
- Ponder2 XML How to parse XML using PonderTalk
- Ponder2 Internals A description of the inner workings of Ponder2
- Ponder2 Tutorial Complete, self-contained runtime files and tutorial with examples
- Glossary Explanation of some of the terms used in this site
- To Do Notes for future changes/fixes

API Documentation

These documents are give detailed documentation of the PonderTalk interface to the standard Ponder2 Managed Objects in PonderDoc format and to the internal Java classes in JavaDoc format.

- Managed Object Documentation Command documentation for Ponder2's core Managed Objects
- Ponder2 Java API Documentation Full internal documentation for Ponder2

Ponder2-Based Systems

- TrPonder A Teleo-Reactive support system using the Ponder2 SMC and Ponder2 Managed Objects
- P2Android - Android integration

Software/Downloads

There are several downloads with different configurations of Ponder2 available.

- Ponder2 Downloads
- Change Log

This software is made available under the terms of the GNU Lesser General Public License as published by the Free Software Foundation.

Tutorials

The following documents contain the current tutorial for learning about Ponder2 by yourself. There are also copies of presentation tutorials given at various conferences in the past.

- Ponder2 Tutorial (Zip file) Complete, self-contained, runtime files and tutorial with examples
- (superseded) UK-Ubinet Workshop (July 2006) *Ponder2.zip* SMCTutorialSlides1up.pdf
- (superseded) Internal DSE tutorial (Nov 2005) *200511 SMC_Tutorial.zip* 200511SMC-AnIntroduction.pdf 200511SMCDocumentation.pdf

Publications

- Ponder2 Publications
- Ponder Publications

Contact Information

This site and the Ponder2 software have been created and are maintained by Dr Kevin Twidle.

For information regarding Ponder2 please email `<ponder2 AT doc DOT ic DOT ac DOT uk>`.

Ponder2Project (last edited 2013-11-25 18:22:14 by KevinTwidle)

Ponder2Acknowledgements

We gratefully acknowledge financial support from the following sponsors towards the development of the concepts, implementation and packaging of the Ponder2 software.

- Allow Project - EU 7th Framework Programme
- Cisco Research Center grant for research into Distributed Systems
- EPSRC AMUSE Project
- IST TrustCoM Project
- IST Emanics Project

Ponder2Acknowledgements (last edited 2008-11-04 15:43:49 by KevinTwidle)

Ponder2Publications

Publications Involving Ponder2

Engineering Policy-Based Ubiquitous Systems

- Morris Sloman and Emil Lupu. The Computer Journal Vol 0, no 0 2009. Paper

Ponder2: A Policy System for Autonomous Pervasive Environments

- Kevin Twidle, Naranker Dulay, Emil Lupu and Morris Sloman. Presented at "The Fifth International Conference on Autonomic and Autonomous Systems" April 2009. ICAS 2009

An Algebra for Integration and Analysis of Ponder2 Policies

- Hang Zhao,Jorge Lobo and Steven M. Bellovin. 9th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2008), 2-4 June 2008, Palisades, New York, USA. Abstract

A Role-Based Infrastructure for the Management of Dynamic Communities

- Alberto Schaeffer-Filho, Emil Lupu, Morris Sloman, Sye-Loong Keoh, Jorge Lobo and Seraphin Calo. Book: *Resilient Networks and Services* pages 1-14. Publisher: *Springer Berlin / Heidelberg* July 2008 Abstract

The Coalition Policy Management Portal for Policy Authoring, Verification, and Deployment

- Brodie, C.; George, D.; Karat, C.-M.; Karat, J.; Lobo, J.; Beigi, M.; Xiping Wang; Calo, S.; Verma, D.; Schaeffer-Filho, A.; Lupu, E.; Sloman, M.. 9th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2008), 2-4 June 2008, Palisades, New York, USA. Abstract

AMUSE: Autonomic Management of Ubiquitous e-Health Systems

- E. Lupu, N. Dulay, M. Sloman, J.Sventek, S. Heeps, S. Strowes, K. Twidle, S.-L. Keoh, A. Schaeffer-Filho. Concurrency and Computation: Practice and Experience, John Wiley and Sons, Inc., 2007 Paper

Towards Supporting Interactions between Self-Managed Cells

- Schaeffer-Filho, A. Lupu, E. Dulay, N. Keoh, S.L., Twidle, K., Sloman, M. Heeps, S., Strowes, S. Sventek, J. First IEEE International Conference on Self-Adaptive and Self-Organizing Systems Boston, Mass., USA, July 9-11, 2007 Paper

Policy-based Management for Body-Sensor Networks

- Keoh, S.L., Twidle K., Pryce, N., Schaeffer-Filho, A E., Lupu, E., Dulay, N., Sloman, M., Heeps, S., Strowes, S., Sventek, J., and Katsiri, E. Proc. 4th International Workshop on Wearable and Implantable Body Sensor Networks (BSN 2007), AAchen, Germany, March 2007 Paper

General SMC Concepts and use of Ponder2 in SMCs

- E. Lupu, N. Dulay, M. Sloman, J.Sventek, S. Heeps, S. Strowes, K. Twidle, S.-L. Keoh, A. Schaeffer-Filho. AMUSE: Autonomic Management of Ubiquitous e-Health Systems. Concurrency and Computation: Practice and Experience, John Wiley and Sons, Inc., 20(3)277-295, 2007. http://dx.doi.org/10.1002/cpe.1194

Authorisation Policies and Conflict Resolution

- G. Russello, C. Dong, and N. Dulay. Authorisation and Conflict Resolution for Hierarchical Domains.IEEE Workshop on Policies for Distributed Systems and Networks 2007 ('p2p-auth.pdf').

Ponder2Publications (last edited 2009-07-29 14:16:41 by KevinTwidle)

Ponder Publications

Ponder2 has been inspired by some of the concepts derived from our experience with Ponder. Here are the main papers that describe Ponder and its applications for network and systems management.

Ponder Basics

- N. Damianou, N. Dulay, E. Lupu and M. Sloman. The Ponder Policy Specification Language. Policy Workshop 2001, Jan. 2001, Bristol, U.K., Springer-Verlag, LNCS 1995.
- N. Dulay, E. Lupu, M. Sloman and N. Damianou. A Policy Deployment Model for the Ponder Language. IFIP/IEEE Symposium on Integrated Network Management, Seattle, USA, 2001, IEEE Press.
- N. Damianou, N. Dulay, E. Lupu, M. Sloman and T. Tonouchi. Policy Tools for Domain Based Distributed Systems Management. IFIP/IEEE Symposium on Network Operations and Management. Florence, Italy, Apr. 2002.

Structuring Concepts, Analysis and Refinement

- E. Lupu, M. Sloman, N. Dulay and N. Damianou. Ponder: Realising Enterprise Viewpoint Concepts. Fourth International Enterprise Distributed Object Computing Conference (EDOC 2000), Makuhari, Japan, Sept. 2000.
- A. Bandara, E. Lupu, A. Russo. Using Event Calculus to Formalise Policy Specification and Analysis. IEEE 4th International Workshop on Policies for Distributed Systems and Networks (Policy 2003). Como, Italy, June 2003.
- A. Bandara, E. Lupu, A. Russo, N. Dulay, M. Sloman, P. Flegkas, M. Charalambides, G. Pavlou. Policy Refinement for DiffServ Quality of Service Management. IEEE eTransactions on Network and Service Management. 3(2):2-13, 2006.

Ponder2Overview

Ponder2 Overview

Contents

Ponder2 comprises a self-contained, stand-alone , general-purpose object management system with message passing between objects. It incorporates an awareness of events and policies and implements a policy execution framework. It has a high-level configuration and control language called PonderTalk and user-extensible managed objects are programmed in Java.

The design and implementation of Ponder2 has been designed to achieve the following goals:

- *Simplicity*. The design of the system must be as simple as possible and incorporate as few built-in elements as possible.
- *Extensibility*. It must be possible to dynamically extend the policy environment with new functionality, to interface with new infrastructure services and to manage new resources.
- *Self-containment*. The policy environment must not rely on the existence of infrastructure services and must contain everything necessary to apply policies to managed resources.
- *Ease-of-use*. The environment must facilitate the use of policies in new environments and prototyping new policy systems for different applications.
- *Interactivity*. It must be possible for managers and developers to simply interact with the policy environment and the managed objects, issue commands to the managed objects and create new policies.
- *Scaleability*. The policy environment must be executable on constrained resources such as Gumstix, PDAs and mobile phones as well as for more traditional distributed systems' management.

Ponder2 can interact with other software and hardware components and is being used in environments ranging from single devices, to personal area networks, ad-hoc networks and distributed systems. Ponder2 is configured and controlled using PonderTalk, a high-level, object orientated language.

Ponder2 implements a self-managed cell (SMC). Management services interact with each other through asynchronous events propagated through a content-based event bus. Policies provide local closed-loop adaptation, managed objects generate events, policies respond and perform management activities on the same set of managed objects. Everything in Ponder2 is a Managed Object. The basic Ponder2 system comprises Event Types, Policies, Domains and External Managed Objects. It is up to the user to create or reuse Managed Objects for other purposes. A Managed Object, including all those mentioned, has to be loaded dynamically into the SMC from a library, thereby producing a factory managed object (c.f. a Java class). The factory managed object can now be sent messages to create new instances of managed objects; these managed objects are the ones which do the work of the system. This is the same as any object oriented system where the class has to be loaded before instances can be created. Once loaded, Ponder2 makes no distinction between factory managed objects and other managed objects. Both types can be sent messages asking them to do something and they both return replies. In the case of the factory managed objects they return a new instance of their underlying type.

Background

Ponder was a highly successful policy environment used by many in both industry and academia. Yet its design suffered from the some of the same disadvantages as existing policy-based frameworks. Their designs were dependent on centralised infrastructure support such as LDAP directories and CIM repositories. The deployment model was often based on centralised provisioning and decision-making. Therefore they did not offer the means for policy execution components to interact with each other, collaborate or federate into larger structures. Policy specification was seen as an off-line activity, and policy frameworks did not allow them to interact easily with the managed systems. Consequently such frameworks were difficult to install, run, and experiment with. Additionally, they usually did not scale to smaller devices as is needed in pervasive systems.

Ponder2 has already been applied in a number of research projects for health monitoring using body-area networks of sensors and actuators [7], unmanned autonomous vehicles [8] as well as large web-service based infrastructures [9]. The software, documentation and tutorials are available from this site.

Ponder2InstallationAndRunning

Ponder2 Installation

Contents

This page describes how to install and run Ponder2 using the `ponder2.jar` file and supporting libraries. You need Java 1.5 or above to run Ponder2. The change log contains details about the various releases and about new, upcoming changes.

Installation is very simple.

1. The Ponder2Downloads page contains the source and binary download files. Download the one that you require. The binary zip file is all that is necessary. The source zip file simply creates the same Jar files that are found in the binary zip, in fact the source zip file was used to create the binary zip.
2. Extract the files with unzip. Use the unzip program on Unix[1], Windows may automatically unzip the file otherwise double-click on it. All the files will be extracted to a directory/folder called *ponder2*

If you have the binary distribution then that's it! You will find an ant build file in the ponder2 directory along with the necessary library Jar files. If you have the source distribution, you need to build the Jar library files as follows:

Step	*Command*	*Description*
1	`cd ponder2/p2src`	Change to the Ponder2 core source directory.
2	`ant install`	Compile the sources and build the Ponder2 and Ponder2Comms library Jar files. You will see some errors concerning `net.ponder2.Shell`, do not worry about these. The Sun Java compiler is sorting out the order of compilation and it comes back to `net.ponder2.Shell` and compiles it successfully later. There are also errors about duplicate `index.html` and `ponder2.css` files, these may be ignored as well. All the binary distribution files will be created and installed in the ponder2 directory.
3	`cd ..`	You now have the same setup as if you had used the Ponder2 binary distribution. The p2src directory is no longer needed unless you want to refer to or change the sources.

Now follow the instructions below to test your installation. Further information regarding interacting with Ponder2 and writing your own Managed Objects can be found from the front page of this site.

Distribution Layout

There are several files and directories at the top level of the distribution, these are:

File/Directory	*Description*
`src/`	This is for your sources. There is an example in `net/ponder/managedobject/SampleObject.java` and `resource/SampleObject.p2`
`lib/`	Directory containing necessary Jar files. Your own Jar files may be added here if required
`doc/`	HTML documentation for the Ponder2 system. There are three sub-directories, described below
`doc/api/`	The JavaDoc API description of all the Ponder2 source files
`doc/pondertalk/`	The PonderTalk documentation for all the Ponder2 Managed Objects
`doc/user/`	The PonderTalk documentation for all the Managed Objects underneath the src directory. The documentation is updated every time the src directory is compiled
`build.xml`	The ant build and run script for simple invocations of the Ponder2 system. By default it compiles the files under the src directory and then runs Ponder2

`p2src/`	The Ponder2 core source files. This optional directory is only present if the source distribution was unzipped. It is not necessary for running Ponder2

Running Ponder2

After downloading the Ponder2 files, Ponder2 can be run as either a stand-alone application or can be started within a Java VM. However it is started, there are some simple tests to ensure that it is running correctly. Ponder2 can be instantiated using Apache Ant, in which case you need Ant version 1.7 or later. If you don't have the appropriate version you can still run Ponder2 by following the instructions for running without ant.

Running with Ant

In the ponder2 directory is an Ant build.xml file. It provides the basic glue to run Ponder2 in a simple manner and should be sufficient for most work. To build and run the basic system simply enter the command

```
ant run
```

If all is well, you will see

```
Shell: trying port 13570
Reading boot.p2
Shell port 13570 ready
```

If you need to run the RMI communications then you can use the folowing command which will give the Ponder2 SMC the address of "`rmi://localhost/Ponder2`":

```
ant run -Drmi=Ponder2
```

The Sun `rmiregistry` program is required to use RMI and the Ant build script will test for its presence and start it in the background if necessary. If Ant has trouble starting `rmiregistry` then it will have to be started by hand first.

In addition to the above commands you may want Ponder2 to read your PonderTalk files so that you can start up your own programs. You can do this with the `-Dboot` option. The argument is a comma separated list of PonderTalk files. For instance to run the example SampleObject PonderTalk file included in the src directory you can use:

```
ant run -Dboot=SampleObject.p2
```

If you want to run a PonderTalk file and then exit you can use the special `exit.p2` PonderTalk file following your own by using a file list e.g.

```
$ ant run -Dboot=SampleObject.p2,exit.p2
...
run:
     [java] Shell: trying port 13570
     [java] Reading boot.p2
     [java] Reading SampleObject.p2
     [java] Adding: A line of text
     [java] Retrieved: A line of text
     [java] Reading exit.p2
$
```

If you want to use Web Services for Ponder2 communications you need a more complex set-up. In this case follow the communications instructions.

Running without Ant

To run the basic system you need `ponder2.jar`, `antlr-runtime.jar` and `qdparser.jar` in your *classpath*. If you need to run the RMI communications or you want to use Web Services for Ponder2 communications you need a more complex setup. In either of these cases follow the communications instructions.

NB All these commands assume that your current directory is the ponder2 directory created by unzipping the distribution file(s).

To run Ponder2 as a stand alone application all that is needed is

Operating System	*Command*
Unix	`java -cp lib/antlr-runtime.jar:lib/qdparser.jar:lib /ponder2.jar net.ponder2.SelfManagedCell [arguments]`
Windows	`java -cp lib\antlr-runtime.jar;lib\qdparser.jar;lib \ponder2.jar net.ponder2.SelfManagedCell [arguments]`

(See startup arguments for argument information)

If all is well, you will see

```
Shell: trying port 13570
Reading boot.p2
Shell port 13570 ready
```

Ponder2 within a Java VM

To start Ponder2 with a Java VM you need to call the main method from a new Thread. To start Ponder2 with a port number of 13570 (the default.) See startup arguments for argument information.

Toggle line numbers

```
1 String args[] = { "-port", "13570" };
2 net.ponder2.SelfManagedCell.main(args);
```

Method calls are available to interact with Ponder2 to create *event types* and *policies* and to inject new events.

Simple Tests

To test Ponder2, open a new terminal session and connect to port 13570 on your computer:

```
$ telnet localhost 13570
```

You will get a shell prompt from Ponder2's built-in, Unix-like shell (but it is not a real Unix shell!). Now enter

```
$ read testsetup.p2
```

This will load and execute the testsetup PonderTalk file which runs some PonderTalk and prints out a few messages. It the defines an event type and a policy which accepts that event. You can now create events causing the policy to be activated. The policy will simply print something out to show that it is working. The policy takes the name of a colour and a value. If the value is over 50 then the policy responds and prints out the event.

You can now try creating an event and sending it into the SMC (Self Managed Cell)

```
$ event colourevent "red" 60
```

This will print out an alert message (in the server window) and shows you that the policy server is running correctly. You should see:

```
$ event colourevent red 23
Command is event
Received event: colour=red intensity=60
```

Use ^C to kill the Ponder2 server. An easier way of starting the policy service so that it reads a PonderTalk file immediately is to use the -boot option. You will still need, however, the Telnet session to be able to enter the `event` command.

```
$ java <java options> -boot testsetup.p2
```

Using Ant you can do the same with

```
$ ant run -Dboot=testsetup.p2
```

Startup arguments

Ponder2 comprises one or more Java jar files depending upon what you want to do. The main Ponder2 jar file (*ponder2.jar*) will pick Managed Objects from other jar files on the *classpath* as necessary. The basic command to start Ponder2 is (assuming that the*classpath* has been set up correctly, including *ponder2.jar*)

```
java [java options including classpath] net.ponder2.SelfManagedCell [Ponder2 options] [ -
[user options]]
```

There are several runtime options that can be passed when Ponder2 is started.

Runtime Option
`Default`
`Examples`
	Multiple Use Allowed?	*Description*

-port <number> `-port 13570`	No	Sets the port number of the Ponder2Shell socket. If <number> is not available and the *multiple* option is used, it is continually incremented until a free port is found. The opened port number is reported to standard output. If the port number is not available and the *multiple* option has not been given, an error is printed on the console. The default port number is 13570. Setting the port to 0 disables it.
-multiple	No	Allows many SMCs to be started on one computer with open shell ports. Ports are tried in numerical succession, starting at the port number given with the *port* option, to find a free one.

-boot <filename> [,<filename>...]
`-boot boot.p2`
`-boot testsetup.p2,exit.p2`
	Yes	Specifies the PonderTalk files to be read when the system starts up. Boot files are evaluated after all other options have been read and acted upon. If there is more than one *filename* or *-boot* option then they are executed in order of declaration. The special *filename* of "-" (i.e. `-boot -`) cancels the loading of all previous boot files including the Ponder2 boot file. Subsequent *-boot* options will still be read

-address <url>
`no default`
`-address rmi://localhost /Ponder2`
`-address socket://4570`
	Yes	Sets the local address of the Ponder2 SMC. If there is more than one *-address* option then Ponder2 adopts all the addresses as its own. The first address becomes its 'favourite,' the protocol it will prefer to send on if it has a choice. Use of this option forces communications protocols to be loaded. It requires one or more other Jar files to be on the class path because the basic *ponder2.jar* file has no communication abilities. See Ponder2Comms for more information

-path <pathname>	Yes	Mounts this Ponder2 SMC into another 'master' SMC as *pathname*. This option is normally used when creating several Ponder2 systems within one single Java VM. The first SMC created becomes the 'master' with the true root domain, the others attach themselves to the main domain hierarchy under the pathname given. **NB** This option is currently disabled.

-auth <allow|deny>
`no default`
`-auth allow`
`-auth deny`
	No	Turns the authorisation system on and sets the default authorisation to *allow* or *deny*. See Ponder2Authorisation for more information about this option.

-version		Prints the version number and exits.
-trace	No	Turns tracing on and prints lots of logging information

CategoryPonder2Installation

1. unzip ponder2.zip (1)

Ponder2Using

Using Ponder2

Once Ponder2 is up and running it will not do much unless it is given something to do. Its built-in Command Interpreter reads and executes PonderTalk which can import new Managed Object types, instantiate Managed Objects, set up Events and Policies and issue commands to previously instantiated Managed Objects.

- PonderTalk introduces the Ponder2 high level language
- Ponder2 Events shows how Events are described inside Ponder2
- Ponder2 Policies gives a breakdown of a Policy description
- Sending Events shows how events are created to trigger Policies
- Executing PonderTalk describes how new PonderTalk commands are given to the SMC at run-timeCommandInterpreter
- Ponder2 XML describes the internal structure of Ponder2 XML

CategoryPonder2Installation CategoryUsingPonder2

Ponder2Using (last edited 2009-05-21 14:12:46 by KevinTwidle)

PonderTalk Overview

PonderTalk is Ponder2's Object Language for configuration and control of the Ponder2 system. Ponder2 has Managed Objects and supports the sending of messages to those objects. This language is a stripped down version of the Smalltalk syntax which has already done the hard work of designing a language for just such a job.

The following is an informal description of the Ponder2 Object Language, the actual grammar file in ANTLR format can be found here.

Contents

General Format

A PonderTalk statement generally comprises a reference to a Managed Object followed by zero, one or more messages to be sent to it. Messages may be simple commands or may contain data to be processed. All message interactions return a result that may be used later on. Ponder2 statements are separated by a full-stop (period) character, like sentences. The final statement does not need one.

Managed objects are referred to by name using the domain hierarchy with the word `root` denoting the top-most domain.

The first line in the example below simply returns the object itself and is often used this way as a message argument.

Toggle line numbers

```
1 object.
2 object message.
3 object message
```

Messages may be sent to the result of a previous message interaction according to precedence rules given below. Parentheses may also be used to give explicit precedence. The following two statements are synonymous, `message2` is sent to the object returned by sending `message1` to `object`:

Toggle line numbers

```
1 object message1 message2.
2 (object message1) message2
```

Many messages may be sent to the same object by `cascading` them; cascaded messages are separated by a semi-colon. Here we see an object receiving three messages, one after another. The results of the first two messages are discarded, the whole statement returns the result of the last message `message3`. The following lines are synonymous:

Toggle line numbers

```
1 object message1; message2; message3
2 object message1. object message2. object message3
```

Assignment

Temporary variables may be created on the fly and may hold the result of a statement. They may be used in place of an object later on.

Toggle line numbers

```
1 temp := root/object.
2 result := temp message.
3 temp another_message
```

Messages

There are three types of messages that can be sent to an object, in all cases the object is mentioned first followed by the message to be sent. Some messages take arguments, arguments are actually statements in their own right and can be quite complex but they mainly comprise other objects and basic types such as strings, numbers or booleans. Other objects will either be named using the domain hierarchy or may be the result of a previous message interaction. Basic types may also be the result of a message interaction or may simply be a "quoted string," a written number or **true** or **false**.

Unary Messages

These messages are single word commands that the object is expected to recognise, they have no arguments and hence carry no additional data e.g. the following example would return a list of the contents of a domain and then activate a policy. (The list would be discarded because it is not used anywhere.)

Toggle line numbers

```
1 root/mydomain list.
2 root/policies/mypolicy activate
```

Binary Messages

Binary messages are generally special characters like "+" and ">>" etc. followed by a single data value. Again, the Managed Object is expected to recognise and understand the message. e.g. the following hypothetical statement could add all the objects in the policy domain to `mydomain`. `Mydomain` would receive the "+" message with the policy domain as the argument and it would take the appropriate action.

Toggle line numbers

```
1 root/mydomain + root/policy
```

Keyword Messages

Keyword messages are much more like other programming language function calls with one or more arguments, the main difference being that each argument is always named in the call. Each keyword may denote the role of the argument.

The following is used to add a Managed Object to a domain. The Managed Object can now be referred to as either `root/mydomain/myobject` or `root/newobj/someobj`.

Toggle line numbers

```
1 root/mydomain add: "myobject" obj: root/newobj/someobj
```

The functions invoked by keyword messages are referred to by concatenating the names of all the arguments, the above is called message `add:obj:`. This is an important thing to know when writing your own Managed Objects in Java.

As mentioned above, arguments may be the result of another message interaction. Here is how a new domain is created and added into the hierarchy. Keyword arguments are often laid out as below for clarity.

Toggle line numbers

```
1 root add: "mynewdomain"
2      obj: root/template/domain create
```

The above code performs the following actions, in order:

1. A new domain is created by sending a `create` unary message to the domain template.
2. The result of that call, the new domain, is handed to the root domain as the `obj:` argument in the `add:obj:` keyword message.
3. This gives us a new domain called `root/mynewdomain`.

Note, it is important to put the quotes to denote a string in the first argument otherwise it would be taken as a managed object and would cause an error, either because the object could not be found or if it were, when the domain receiving the message finds that it is not a string.

Why were things done in this order? Why was the `create` message sent first? This brings us to message precedence.

Message Precedence

The above example works because of the precedence given to messages. Normally everything proceeds left to right, however if there is a mixture of message types within a statement the precedence rules take over. Unary messages are performed first, followed by binary messages followed by keyword messages. If any other precedence is required then parentheses must be used in the normal fashion.

Following on from the above example. If we wanted to put the domain template into another domain and *then* use it to create a new domain we would have to do one of the following:

Toggle line numbers

```
1 root add: mydomaintemplate obj: root/template/domain.
2 root/mydomaintemplate create
```

Using the fact that a domain `add:obj:` message returns the Managed Object just added, we can do:

Toggle line numbers

```
1 (root add: mydomaintemplate obj: root/template/domain) create
```

Of course, in the above two examples we have just lost track of the new domain that was returned because we didn't put it anywhere.

Further Reading

More information about PonderTalk can be found in:

- PonderTalk Syntax
- Basic PonderTalk types

PonderTalkOverview (last edited 2009-07-27 15:06:32 by KevinTwidle)

PonderTalk Syntax

Contents

PonderTalk is the high-level language, based on Smalltalk, that is used to control and interact with the Ponder2 SMC. Ponder2 managed objects are very similar to Smalltalk classes insofar as they can be created (in our case imported), instantiated and can be sent messages which return replies. The main difference being that Managed Object are written in Java and instances of managed Objects may be held in a remote SMC, messages are sent correctly regardless of the location of the Managed Object.

PonderTalk	*Smalltalk*	*Description*
Managed Object	Class	Ponder2 Managed Objects are the equivalent of Smalltalk Classes
Managed Objects written in Java	Class Objects written in Smalltalk	Managed Objects are "imported" into an SMC. The Smalltalk syntax to create new Classes has not been included in PonderTalk
Managed Objects may be local or remote	Class instances are always local	It is not necessary to know whether a Managed Object is local or remote, Ponder2 is a transparent, distributed system

The Language

PonderTalk statements are like sentences separated by a "." PonderTalk statements may either be an assignment to a temporary variable or may send a command to a Managed Object. The basic syntax is:

Toggle line numbers

```
var := something.
managed_object command.
```

All commands return a value so we can have:

Toggle line numbers

```
// Create a new domain, assign it to 'var1' and 'var2'
var1 := var2 := newdomain create.

// Put 'clock' into the new domain and call it myobj
var1 at: "myobj" put: clock.

// Get myobj from the new domain, using var2, and assign it to var3
var3 := var2 at: "myobj".

// Perform some operation on the clock object
var3 sendTime
```

Commands have three forms, one with no arguments, one with one argument and one with one or more arguments. These three forms are UnaryMessages, BinaryMessages and KeywordMessages.

UnaryMessages

Unary messages are simply commands in the form of a word sent to a Managed Object. A unary message has no associated data with it.

Examples of Unary messages include

Toggle line numbers

```
// Activate a policy
root/policy/pol1 activate.

// Create a new domain
root/factory/domain create.
```

Binary Messages

Binary messages comprise one or more symbols with a single argument. It is entirely up to the object receiving the binary message what it does with it.

Examples of Binary messages include

Toggle line numbers

```
"String1" + "String2".

// Some binary messages can look bizzare
// The following is the binary operation "%^&$%"
root/dom/myobj %^&$% root/someotherobj.
```

Keyword Messages

Keyword messages are more like traditional method or function calls except that all arguments are named arguments. e.g. the following adds an object to a domain, giving it the name myObj

Toggle line numbers

```
/root/dom1 at: "myObj" put: newObject.
```

This can be read as calling the method `dom1.at_put_("myObj", newObject)`

Precedence

Unary messages take precedence over binary messages which take precedence over keyword messages. Assignment has the lowest precedence. If there are equal precident message types then the precidence rule is left to right. Parentheses can be used to overide presidence and to separate keywords from different commands getting mixed together. The following table show examples of statements involving a mixture of commands. Each statement is repeated with parentheses showing the same statement taking precedence into account.

As written	Meaning	Description
`myObj command1 command2`	`(myObj command1) command2`	The return value of the unary message `myObj command1` is sent the unary message `command2`
`myObj ucmd1 + "hello"`	`(myObj ucmd1) + "hello"`	The return value of the unary message `myObj ucmd1` is sent the binary message `+ "hello"`
`myObj + "hello" length`	`myObj + ("hello" length)`	The unary message `length` is sent to the string, the result of that is sent as part of the binary message `+` to `myObj`
`myObj cmnd1 key1: arg1 key2: arg2`	`(myObj cmnd1) key1: arg1 key2: arg2`	The result of sending `cmnd1` to `myObj` is send the keyword message `key1:key2:` with the arguments `arg1` and `arg2`
`myObj at: "name" length put: obj`	`myObj at: ("name" length) put: obj`	The result of `"name" length` becomes the first argument in the keyword message `at:put:`

`myObj + 3 * 5`	`(myObj + 3) * 5`	Left to right if equal precidence, all binary operators are equal

`myObj at: "fred" put: newObj at: "fred"`	???	Seen as operation `at:put:at:`, probably not what is wanted but could be valid

Special Types

In addition to managed objects being treated as PonderTalk objects and being used as arguments and receivers of messages, there are several built in object types.

Blocks

Blocks are sections of one or more statements with an optional set of arguments that may be executed later, rather like a stored procedure. Blocks take the following form:

Toggle line numbers

```
1 [ :arg1 :arg2 | statement1. statement2 ]
```

Blocks are typically executed by sending them the unary message `value` if there are no arguments to be filled in If there are arguments than keyword messages are used, they take the form `value:`, `value:value:` or `value:value:value:` etc, one keyword for each argument in the block. There are other ways to execute a block but these are included in the block documentation. For example the following would print "Hello, World!"

Toggle line numbers

```
1 a := [ root print: "Hello, World!" ].
2 a value
3
4 Hello, World!
```

Further Reading

Now that you have learnt the syntax, you can see the basic types that PonderTalk makes available.

PonderTalkSyntax (last edited 2009-09-16 13:47:35 by KevinTwidle)

PonderTalk Basic Types

In addition to Managed Objects that have a command interface there are several basic, built-in objects that can be manipulated and given to Managed Objects as arguments. They are:

Name	*Description*
Hash	A collection of named objects c.f. Perl hash, Java Map, Smalltalk dictionary
Number	An integer or real number
Array	An array of other objects
String	A string value
Boolean	A true or false value
Nil	The null value
Xml	An XML structure

Some of the following examples make use of special operations available on the root domain of the SMC. These include import and print, please see the root domain documentation for these operations.

String

Strings are created in PonderTalk with double-quotes e.g.

Toggle line numbers

```
1 myString := "A string"
```

Operations

Operation	*Result Type*	*Description*	*Example*
+ object	String	Returns a new string being the concatenation of the original string and the object as a string	"S1" + "S2" => "S1S2"
* number	String	Returns a new string being the concatenation of `number` copies of the original string	"S1" * 3 => "S1S1S1"

More operations to do sub-strings and regexp will be added. If an operation is expecting a Boolean then the String value "true" becomes true and the String value "false" becomes false otherwise an error is created. See the string documentation for all operations available.

Block

A block is an object that contains code. It behaves rather like a function with zero or more arguments. When a block is executed it returns the value of the last statement executed. The result of the last statement becomes the result of the block. Blocks may be created in PonderTalk with square brackets e.g.

Toggle line numbers

```
1 // A block with no arguments, this will return 7
2 [ myNum := 3 * 4 ]
3 // A block with two arguments, this will return the result of arg1 + arg2
4 [ :arg1 :arg2 | arg1 + arg2 ]
```

Blocks are executed by sending them value messages. `value` if there are no arguments, `value:` if there is one argument, `value:value:` for two arguments etc. etc.

Operations

Operation	*Result Type*	*Description*	*Example*
value	Object	Executes the statements inside the block	["Hello,"] **value** => "Hello,"
value: arg	Object	Executes the statements inside the block with `arg` being given as the block's argument	[:arg | "Hello," + arg] **value:** "World!" => "Hello, World!"
value: arg1 **value:** arg2	Object	Executes the statements with `arg1` and `arg2` being given as the block's arguments	[:arg1 :arg2 | "Hello," * arg1 + arg2] **value:** 2 **value:** "World!" => "Hello, Hello, World"

See the block documentation for all operations available.

Boolean

Booleans are created in PonderTalk with the special keywords **true** and **false** e.g.

Toggle line numbers

```
1 myBoolean := true.
2 myBoolean := false.
```

Operations

Operation	*Result Type*	*Description*	*Example*
ifTrue: Block	answer the result of the block or Nil	Execute the block (with no parameters) if the boolean is true	myBoolean **ifTrue:** [root print "true"]
ifFalse: Block	answer the result of the block or Nil	Execute the block (with no parameters) if the boolean is false	myBoolean **ifFalse:** [root print "false"]
ifTrue: BlockT **ifFalse:** BlockF	answer the result of one of the blocks	Execute BlockT if the boolean is true else BlockF. Neither block will receive arguments	myBoolean **ifTrue:** [root print "true"] **ifFalse:** [root print "false"]
ifFalse: BlockF **ifTrue:** BlockT	answer the result of one of the blocks	Execute BlockT if the boolean is true else BlockF. Neither block will receive arguments	myBoolean **ifFalse:** [root print "false"] **ifTrue:** [root print "true"]
not	Boolean	returns the inverse of the boolean value	myBoolean := myBoolean not
& Boolean	Boolean	return the value of the two booleans joined with and	myBoolean := boolean1 **&** true
| Boolean	Boolean	return the value of the two booleans joined with or	myBoolean := boolean1 | true
and: aBlock	Boolean	Return false if the boolean is false else return result of the block. The block is not executed if the boolean is false	myBoolean **and:** [4 < 5]
or: aBlock	Boolean	Return false if the boolean is false else return result of the block. The block is not executed if the boolean is true	myBoolean **or:** [4 < 5]

See the boolean documentation for all operations available.

Array

Arrays hold an ordered collection of Managed Objects. The contained objects are not typed. Arrays are created from

other operations like `collect:`. They may also be created with PonderTalk in the following manner:

Toggle line numbers

```
1 // Create an empty array
2 anArray := #().
3 // Create an array with a variety of elements
4 array := #( 5 "20" root/factory/ecapolicy true ).
5 // Obtain an array as a result of a messag
6 anotherArray := root/factory list.
```

Operations

Operation	*Result Type*	*Description*	*Example*
at: aNumber	Object	Return the object at index aNumber	myObj := root **at:** 5
do: Block	self	For each object call the block with the object as an argument. Only the last block's statement return value is returned	#(1 2 3) **do:** [:value | account **add:** value]
collect: Block	Array	For each object in the arraycall the block with the the object as an argument. An array of all the block's return values is returned	array := #(1 2 3 4 5 6 7 8 9 10). pages := array **collect:** [:value | book **at:** value].

See the array documentation for all operations available.

Hash

Hashs are created in PonderTalk from other operations. Domains and Events are actually Hashs

Toggle line numbers

```
1 // Use an array to create an empty hash
2 aHash := #() asHash.
3 // or a domain
4 anotherHash := root/factory asHash.
```

Operations

Operation	*Result Type*	*Description*	*Example*
at: "name"	an object	Return the object at entry "name"	myObj := root **at:** policy
at: "name" **put:** Object	Object	Put object into the hash and call it "name" and return the object	root at: "fred" **put:** myObj
do: Block	value of last block executed	For each name/value pair call the block with the two arguments. Only the last block's statement return value is returned	root **do:** [:name :value | root **print:** name]
collect: Block	Array	For each name/value pair call the block with the two arguments. An array of all the blocks' return values is returned	allObjs := root **collect:** [:name :object | object]. allNames := root **collect:** [:name :object | name].

See the hash documentation for all operations available.

Xml

The Xml basic type holds an XML structure. The structure may be manipulated using XPath commands sent to the structure as PonderTalk messages. The Xml type is created from a String using the `asXML` unary message. Note: In the

following example single quote characters (') have been used within the XML. This is valid XML and means that double quote characters (") do not have to be escaped.

Toggle line numbers

```
1 // Create an Xml basic object
2 xml := "<element attr1='a1' ><child>Some text</child></element>" asXML.
```

Operations

Operation	*Result Type*	*Description*	*Example*
xpathNumber: "expression"	number	Return a number from the structure	num := xml **xpathNumber:** "@attr1"
xpathString: "expression"	string	Return a string from the structure	string := xml **xpathString:** "@attr1"
xpathNode: "expression"	Xml	Return xml struct from the structure	xml1 := xml **xpathNode:** "child"
xpathNodeSet: "expression"	Array	Return xml structs from the structure	array := xml **xpathNodeSet:** "childName"

Toggle line numbers

```
1 // Handle incoming events
2 eventHandler := [ :string |
3     root print: "New event received from ISL".
4     root print: string.
5
6     // Convert to an XML object
7         xml := string asXML.
8
9         topic := xml xpathString: "/esiievent/@topic".
10        event := xml xpathNode: "/esiievent/event".
11        //root print: "Event is " + event.
12        eventName := event xpathString: "@name".
13        root print: "topic is " + topic + " " + eventName.
14
15        response := event xpathString: "response/@key".
16        root print: "response is " + response.
17
18        hash := getAttributes value: event .
19        hash at: "response" put: response.
20
21        root print: "Sending event "+eventName+" internally".
22        (root/esii/event at: eventName) fromHash: hash.
23
24 ].
25 root/esii at: "eventHandler" put: eventHandler.
```

See the XML documentation for all operations available.

PonderTalkBasicTypes (last edited 2008-07-16 21:31:14 by KevinTwidle)

Ponder2 Events

Events are generated from Event Templates which describe the attribute values that a particular event should carry with it. An Event Template is created from the Event Factory. An Event Template is the specification of a particular event with named arguments. Instances of the Event Template are created and then picked up by Policies that have "subscribed" to that particular Event Template.

The Event Factory is generally located at */factory/event*. An Event Template is created by sending the event factory a create command with an array of attribute names that the event will carry.

An event can be generated using PonderTalk by sending the event template a create command with an array of values, one per expected attribute, in the correct order. A Java Managed object can also create events using the operation method on the event template managed object.

Example

To create an Event Template called */event/testevent* that has two arguments, colour and intensity, we can use the following PonderTalk:

Toggle line numbers

```
1 template := root/factory/event create: #( "colour" "intensity" )
2 root at: "colourevent" put: template.
```

We can now create and send an event of this type. If sending an event using PonderTalk you can do one of the following:

Toggle line numbers

```
1 // If the variable "template" is still in context.
2 template create: #( "red" 35 ).
3
4 // Can be used at any time
5 root/colourevent create: #( "red" 35 ).
```

This will create an event and dispatch it into the system where it will be handed to the appropriate policies.

As can be seen in the documentation at http://www.ponder2.net/doc/pondertalk/EventTemplate.html an event can be created from a Hash. The following examples generate the same event as the one above.

Toggle line numbers

```
1 template fromHash: #( "intensity" 35 "colour" "red" ) asHash.
```

Toggle line numbers

```
1 hash := #() asHash.
2 hash at: "colour" put: "red".
3 hash at: "intensity" put: 35.
4 template fromHash: hash.
```

If creating an event from a Managed Object, the `create` message is sent in the normal way:

Toggle line numbers

```
1 @Ponder2op("mkEvent:")
2 protected void makeEvent(P2Object source, P2Object anEventTemplate) {
3     anEventTemplate.operation(source, "create:",
4       P2Object.create(P2Object.create("red"), P2Object.create(35)));
5 }
```

If the above object has been instantiated as **myObj**, the following PonderTalk can be used:

Toggle line numbers

```
1 myObj mkEvent: root/colourevent.
```

This will create an event and dispatch it into the system where it will be handed to the appropriate policies.

Note: For the event to be propagated properly, the managed object (**myObj** in this case) *must* be a member of a domain in the domain hierarchy. See Ponder2EventBus for more details.

Other methods of introducing events into the system are described in Ponder2SendingEvents.

CategoryPonder2Project

Ponder2Events (last edited 2008-08-05 17:05:11 by KevinTwidle)

Ponder2EventBus

Ponder2 Proximity Event Bus

Ponder2 has an unusual graph-directed event mechanism. A normal event bus can be described as a distributed computer system in which a plurality of applications are connected to an event bus through an adapter interface. Messages exchanged between applications are transmitted on the event bus and are processed according to rules stored in a rules repository. This type of event bus distributes all events to all objects/processes that have registered to receive it. In our case, events are distributed to policies that have specified that they want to receive that particular event type. However, it is not always useful for all policies to be activated by the same event so extra processing is required to determine if it is required or not. This results in extra complexity for the policy or a proliferation of event types to give a finer graduation of events.

Ponder2 solves this problem by the use of a Proximity Event Bus which allows policies to pick up events generated only by certain managed objects. Essentially, instead of “attaching” policies to the event bus when they are activated, they may explicitly be attached to managed objects within the domain hierarchy. That is they subscribe to an event type via a managed object rather than via the global event bus. An event is always produced by a managed object somewhere within the system. Instead of the event being offered to all the policies in the system that are dependent on that event, the event propagates up the domain hierarchy. As it is propagated it is offered to any policies attached to any domain on the path up to the root as long as the event type is expected by those policies. In this way, a policy can be set up that will only respond to events created by a limited set of managed objects.

The more traditional Event Bus can be emulated simply by attaching all policies to the root domain meaning that all events are offered to all policies.

{{attachment:ProxyEventBus.jpg||height="361",width="406"|height="494",width="555"}}

As can be seen in the above figure there are two identical bed stations, each with a heart rate monitor which will produce a “heart rate” event should the heart rate cause concern. The bed policies and the ward policy have all subscribed to the heart rate event. With a traditional event bus, the bed policies would have to check where the event came from before proceeding. With the Proximity Event Bus the bed policies will only see events generated from managed objects within their bed domains and the ward policy would see all events. The bed policies’ actions could involve just local managed objects and therefore they don’t need any special knowledge about which bed they are monitoring. This allows more bed systems to be introduced within the ward without configuration and creating minimum impact on the rest of the system.

The Proximity Event Bus becomes even more relevant when applied to distributed systems allowing local policies to deal with the local events while maintaining the ability to have higher-level policies reacting to the same event from many places.

Note: Objects creating event *must* be held in the domain structure otherwise the event will not be propagated.

Ponder2EventBus (last edited 2008-08-01 16:15:18 by KevinTwidle)

Ponder2SendingEvents

Sending Ponder2 Events

Once an Event Template has been created, an event can be generated from a stand-alone application, a direct Java call, a Java RMI call, a Web Service call, from directly executed SMC XML or from the command-line shell.

Example Event

The following descriptions of sending events use this event definition. The event contains a *colour* and an *intensity* as attributes.

Toggle line numbers

```
1 template := root/factory/event create: #( "colour" "intensity" )
2 root/event at: "colourevent" put: template.
```

Stand-Alone Application

An event can be generated in the SMC using a simple application embedded within the Ponder2 Jar file. This application uses RMI to make a call to the receiving SMC and it sends an arbitrary PonderTalk string. The execution takes the format:

```
java -classpath ponder2.jar net.ponder2.PonderTalk RMIname PonderTalk statements.
```

where ... indicates many Jar files and one or more optional arguments. You must have as many arguments as the event in question was defined with. This example expects that a PonderTalk managed object has been started and added to the domain hierarchy.

```
java -classpath ponder2.jar net.ponder2.PonderTalk //localhost/MyPonder2 'root/event
/colourevent #( "red" 53 ).'
```

The result will be printed out. If something goes wrong then the command will exit with a return code of 1 and will print a short error message. Remember to escape (\") the quotes if you are using a Unix shell.

Java Call

If the SMC is running in the same VM as your Java application then you can create the above event with a direct method call again using PonderTalk:

Toggle line numbers

```
1 String p2xml = P2Compiler.parse("root/event/colourevent create: #( \"red\" 53 ).");
2 return new XMLParser().execute(SelfManagedCell.RootDomain, p2xml);
```

The first argument to `execute()` is the source or creator managed object of the compiled XML i.e. the object that is sending the create message to the `colourevent` managed object. The source managed object is needed by event creation because the event is actually sent by that object. If you don't have an object that you want to use to send the event then the root domain `SelfManagedCell.RootDomain` may be specified, as in the example above.

Java RMI

If the SMC is running within another VM on the same or different machine then RMI may be used to send an event. This method uses the PonderTalk managed object which will compile and execute PonderTalk. If there is anything wrong an exception will be thrown.

First the target SMC must create an instance of a PonderTalk managed object with an RMI name to be registered (we assume that *rmiregistry* is already running):

Toggle line numbers

```
pondertalk := root load: "PonderTalk".
root/factory at: PonderTalk" put: pondertalk.

// Create an instance waiting on the RMI port MyPonder2
pondertalk := root/factory/PonderTalk create: "MyPonder2".
// The managed object *must* be put into a domain otherwise events will not be sent
root at: "pondertalk" put: pondertalk.
```

To create the test event described above, the following Java code snippet can be used:

Toggle line numbers

```
import net.ponder2.Ponder2Interface;

...
String rmiName = "//localhost/MyPonder2";
String ponderTalk =  "root/event/colourevent create: #( \"red\" 53 ).";
String result;
try {
    PonderTalkInterface pt = (PonderTalkInterface)Naming.lookup(rmiName);
    result = pt.execute(ponderTalk);
}
catch (Exception e) {
    System.out.println("An error has occurred: " + e.getMessage());
}
```

The RMI method definition is

Toggle line numbers

```
public String execute(String ponderTalk) throws RemoteException;
```

Web Service

<This section is out of date >

Ponder2 can have an event sent to it using a Web Service call. The arguments are the port number that the SMC was started at, the event name and a sequence of string arguments. See the WSDL for more information.

```
<wsdl:message name="eventRequest">
   <wsdl:part name="port" type="xsd:int"/>
   <wsdl:part name="eventName" type="xsd:string"/>
   <wsdl:part name="args" type="impl:ArrayOf_xsd_string"/>
</wsdl:message>
```

Command Line Shell

After starting the SMC and observing the port number that it has started at, you can telnet to the SMC and access its shell command module. The following SMC output shows it starting at socket 13571

```
Shell: trying port 13570
Shell: trying port 13571
Shell port 13571 ready
```

So, to access this SMC enter the following telnet command in a new terminal window:

```
$ telnet localhost 13571
```

The shell may look a little like a Unix shell but it is not as sophisticated. You can now create events by typing (to the SMC shell)

```
$ event root/event/colourevent red 53
```

As a shorthand simplification the `root/event/` may be omitted, however if the event type is in any other domain, the full path name must be used. You can only send simple arguments without spaces using this method. It is useful for quick testing.

For more complex arguments, PonderTalk can be used:

```
$ root/event/colourevent create: #( "red" 53 ).
```

CategoryPonder2Project CategoryUsingPonder2

Ponder2SendingEvents (last edited 2009-02-12 17:35:55 by KevinTwidle)

Ponder2 XML

Ponder2 has an XML managed object which holds XML and can apply XPath queries on it. The XML is a Ponder2 basic managed object and therefore does not need to be loaded. An XML managed object can be created from a String by sending it the "asXML" unary message.

Toggle line numbers

```
1 xml := "<html><head/><body>Hello, World!</body></html>" asXML.
```

The following messages are accepted by the XML managed object:

Operation	*Return*	*Description*
`xpathNodeSet: anXPathExpression`	Array	Answers an array with XML elements generated by applying anXPathExpression to the receiver
`xpathNode: anXPathExpression`	XML	Answers an XML element generated by applying anXPathExpression to the receiver
`xpathString: anXPathExpression`	String	Answers a string generated by applying anXPathExpression to the receiver
`xpathBoolean: anXPathExpression`	boolean	Answers a boolean generated by applying anXPathExpression to the receiver
`xpathNumber: anXPathExpression`	int	Answers a number generated by applying anXPathExpression to the receiver
`asString`	String	
`asXML`	XML	Returns itself, used for compatibility with other types

Given the following XML representation of a Ponder2 event, we will proceed to parse it into a hash managed object:

Toggle line numbers

```
1 <?xml version='1.0' encoding='UTF-8'?>
2 <p2event name='testevent'>
3     <variable type='Hash'>
4         <element name='phone'>
5             <variable type='String'>1234</variable>
6         </element>
7         <element name='address'>
8             <variable type='String'>London</variable>
9         </element>
10        <element name='name'>
11            <variable type='String'>Fred</variable>
12        </element>
13    </variable>
14 </p2event>
```

First we have to create the hash and convert the string containing the XML into an XML managed object.

We then pick up the `p2event` element using XPath, this returns another XML managed object with the focus on the `p2event` element, from now on we can use relative XPath expressions.

Toggle line numbers

```
1         hash := #() asHash.
2
3         xml := string asXML.
4
5         event := xml xpathNode: "/p2event".
```

```

        eventName := event xpathString: "@name".
        root print: "event name is " + eventName.

        // Get the 'variable' element
        var := event xpathNode: "variable".

        // Get an array of 'element' nodes
        elements := var xpathNodeSet: "element".
        elements do: [ :xml |
            // Get the name= attribute
            name := xml xpathString: "@name".
            // get the element data
            value := xml xpathString: ".".
            root print: "Setting " + name + " to " + value.
            hash at: name put: value.
        ].
```

We can now get the name of the event from the attribute followed by the `variable` structure which contains the names and values of the variables in the hash.

Since we have an array of elements we can cycle through them, taking the name and value from each, and add them to the hash. Note that although the block is a closure and it cannot affect the values of variables outside its context, it can affect the contents of other managed objects. So, in this case it can change the contents of the has because `hash` is a reference to the actual managed object.

We now have in the hash a proper representation of the XML structure. This structure while being simple is a little more complicated than is required because it fits in with the xmlBlaster example.

Ponder2XML (last edited 2008-09-23 13:03:09 by KevinTwidle)

Ponder2Eclipse

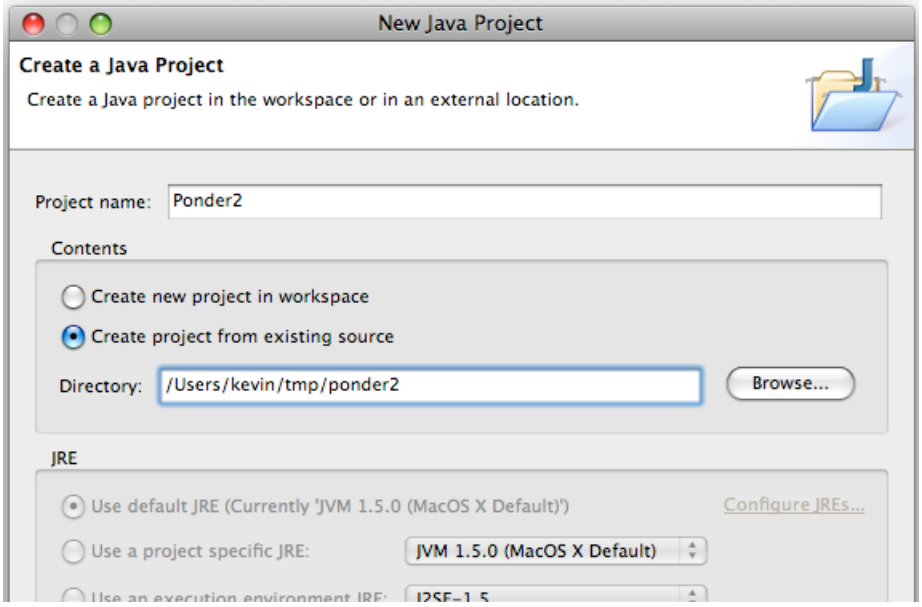
Using Eclipse for Ponder2 Development

Eclipse provides an excellent development environment for Ponder2. This page contains instructions for setting up Eclipse to use Sun's Java APT compiler which is necessary for compiling Ponder2 Managed Objects and generating the stub-code required for Ponder2.

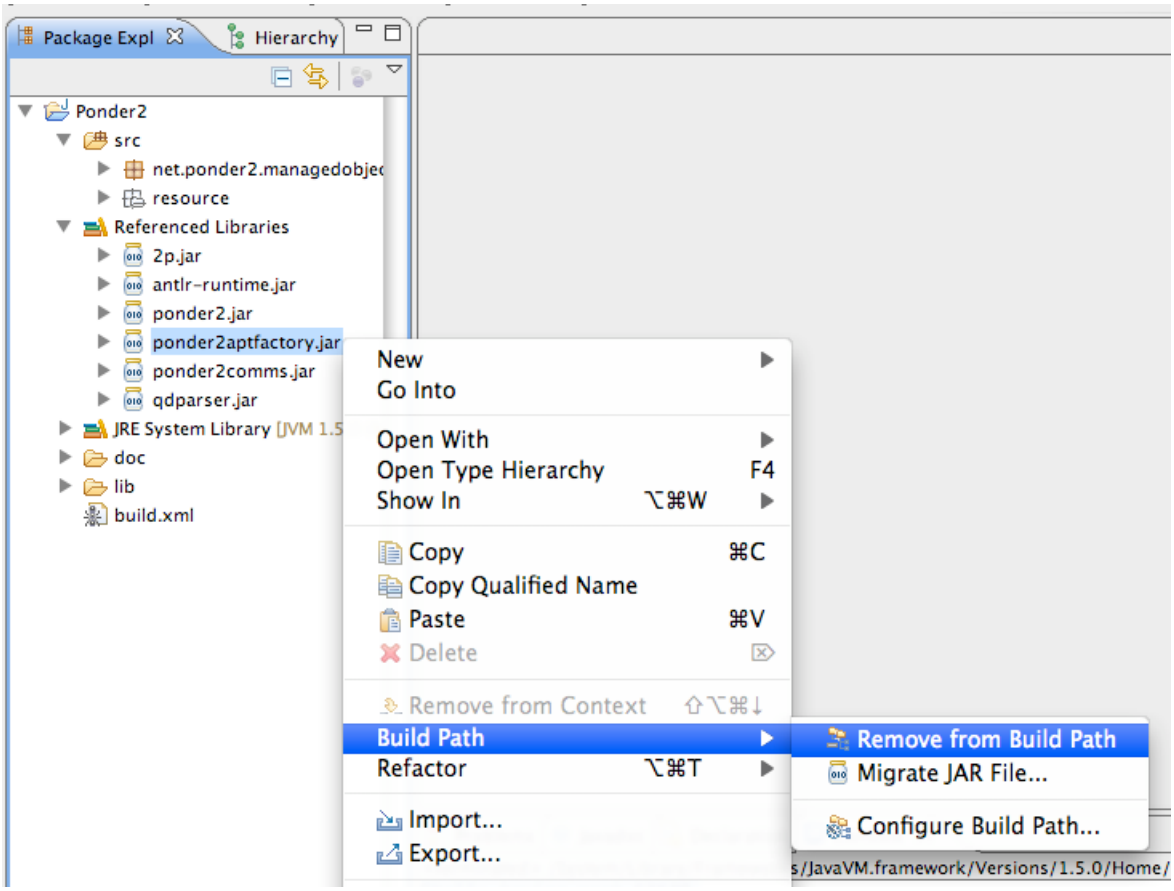
Contents

Assuming that you have the latest release of Ponder2 in a directory called `ponder2` it is easy to import it as a new project in Eclipse:

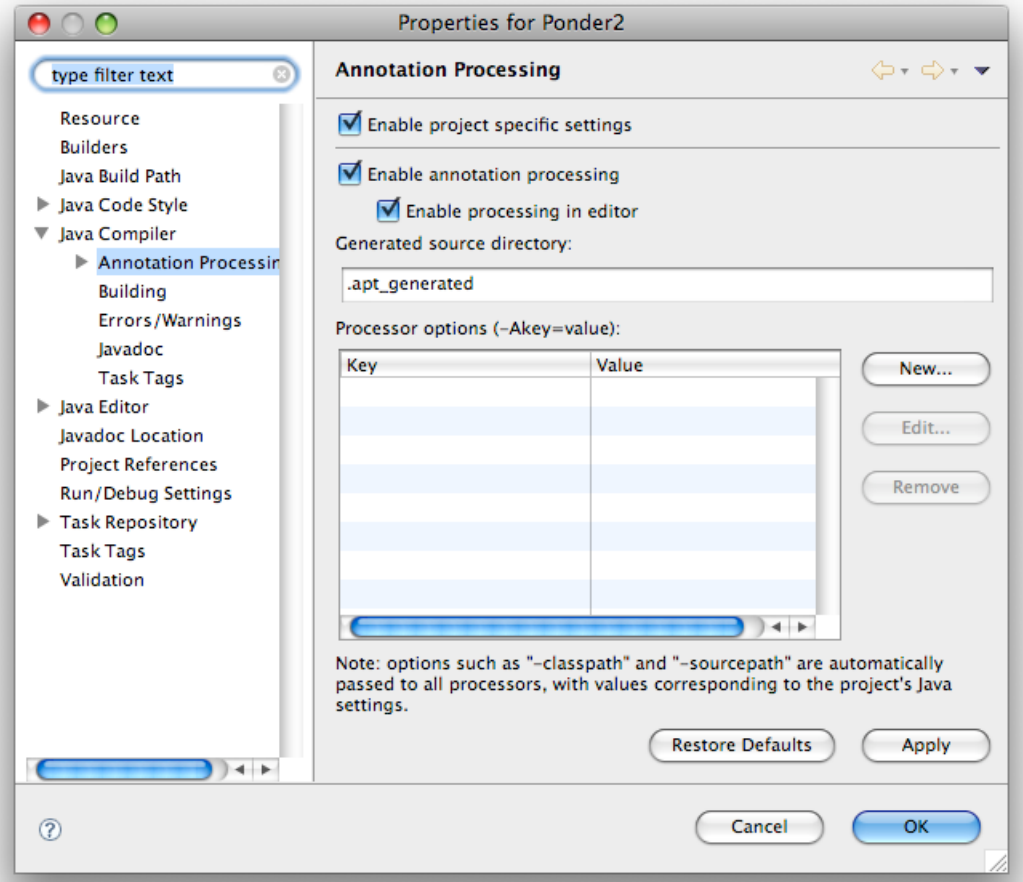
1. Download either the source or binary release of Ponder2
2. If you have the source release then build the Ponder2 system by going into the ponder2 directory and typing `ant build`
3. Open Eclipse
4. Create a new Java Project



5. Give the project a name e.g. Ponder2
6. Select "Create project from existing source" and select the ponder2 directory you created
7. Expand the project
8. Remove the ponder2aptfactory.jar file (if present) from the build path

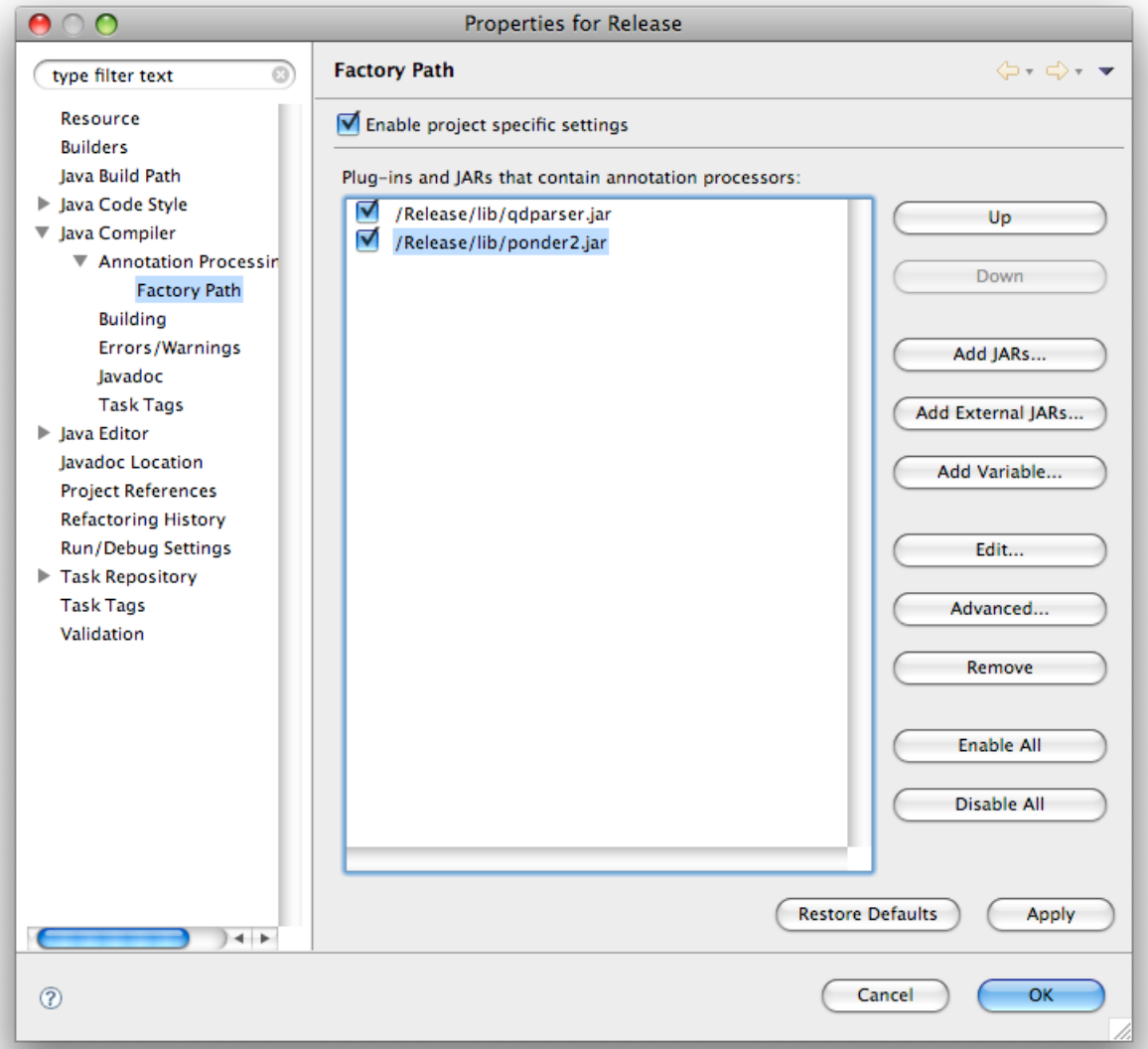


9. Select the project and right-click to get the preferences/properties for the project
10. Select Java Compiler | Annotation Processing and enable project specific settings and enable annotation processing

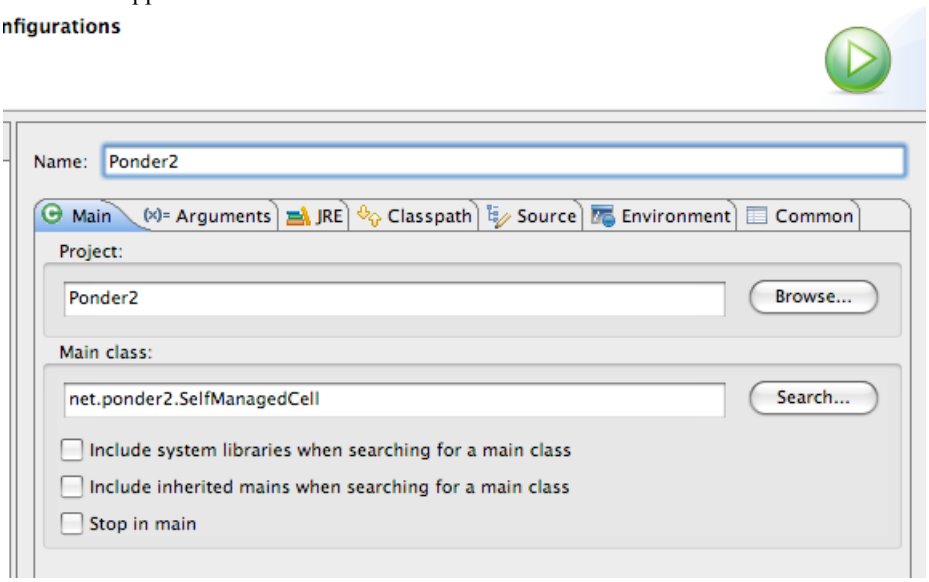


11. Open up the **Annotation Processing** triangle, select **Factory Path**, enable project specific settings and add the

two necessary Jar files (`qdparser.jar` and `ponder2.jar`) from your lib directory as as shown. Use the **Add Jars...** button and locate your project and the lib directory. The project name `Release` will differ in your Eclipse setup

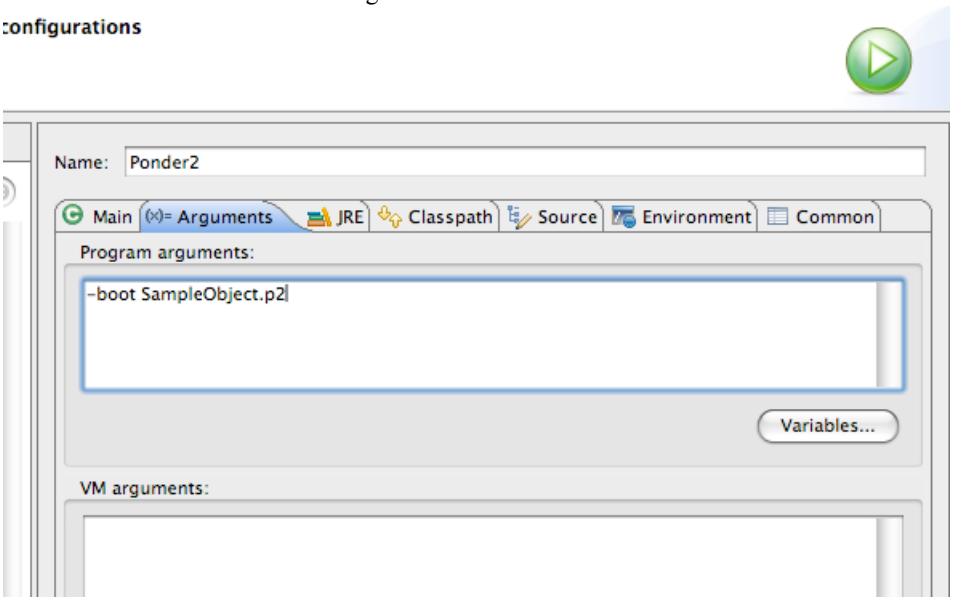


12. Click OK to close the window
13. Now run the Ponder2 sample object test
14. Open the run configurations dialog
15. Select new application



16. Give it a name e.g. SampleObject

17. search for a main - select SelfManagedCell from the list



18. Select arguments and add `-boot SampleObject.p2`
19. Click run
20. You should see the following output in the console window

```
Shell: trying port 13570
Reading boot.p2
Reading SampleObject.p2
Adding: A line of text
Retrieved: A line of text
Shell port 13570 ready
```

Ponder2Eclipse (last edited 2008-09-16 18:45:38 by KevinTwidle)

ManagedObjectsSimple

Managed Objects

Contents

A Managed Object is an entity in Ponder2 capable of receiving and replying to PonderTalk messages. A Managed Object is written in Java and uses Java annotations (i.e. `@Ponder2op()`) to create the links between PonderTalk message keywords and Java methods. This page gives an example of a Managed Object's design and implementation. After reading this you may like to progress to the more advanced guide.

AlarmClock

For this example we will design a Managed Object that can show and hide a picture of an alarm clock. In addition, it will be able to set the alarm clock ringing and turn it off again. We will use the GIF images below for the clock in its quiescent and ringing states. The window handling will be simply ordinary Java swing programming.

Operational Commands

The first thing to do is to decide on the name and the interface of your managed object. Let's call the new object AlarmClock. We will be controlling our managed object using PonderTalk. We need to show and hide the alarm clock and we need to be able to turn the alarm on and off. Assuming that our instantiated managed object is called `alarm`, the following PonderTalk commands will do the trick.

```
// Show the window
alarm show.

// Hide the window
alarm hide.

// Set the alarm ringing
alarm setAlarm.

// Cancel the alarm
alarm cancelAlarm.

// Set the alarm ringing and show it
alarm setAlarm show.
```

It would be nice to get the status of the alarm as well in case we need to test it. The following would return a boolean

```
// Is the alarm visible? Returns a boolean
alarm isVisible.

// Is the alarm on? Returns a boolean
alarm isAlarmOn.
```

Well that is the basic functionality but the above commands are a little add-hoc, they can be improved and made to be more consistent by combining them as follows

```
// Show or hide the window
alarm visible: <true | false>.

// Is the window shown or hidden, returns a boolean
alarm visible.

// Set or cancel the alarm
alarm alarm: <true | false>.
```

```
// Is the alarm on?
alarm alarm.
```

More formally this gives us:

Command	Argument(s)	Return	Description
visible:	aBoolean	self	Shows the alarm window if aBoolean is true else hides it
visible		aBoolean	Returns true if the alarm window is visible else false
alarm:	aBoolean	self	Sets the alarm ringing if aBoolean is true else stops it
alarm		aBoolean	Returns true if the alarm is ringing else false

This style of command choices works out quite well i.e. use `something: arg` to set an attribute and `something` to return the current value. So, we will have four separate commands that the alarm managed object will accept.

Factory Commands

We also need to know how our managed object it to be created from its factory object. Normally a simple `create` message will suffice but in this case we should give the window a title. Since the title will not change during the life of the managed object it is appropriate to give the title when the object is instantiated.

```
// Create an instance of AlarmClock
factory := root import: "AlarmClock".
alarm := factory create: "A title".
```

This gives us:

Command	Argument(s)	Description
create:	aString	Return a new instance of AlarmClock with the title set to aString

Directory Setup

If you have your own Java development environment you can keep your sources in that and issue the appropriate command(s) to compile and run your alarm clock. However, if you follow these instructions you can use the supplied ant `build.xml` file to create and run your managed objects. In your Ponder2 directory you will have the following directories and files:

```
build.xml
lib/
p2src/ (optional)
src/
```

You will put your files underneath `src`:

```
src/
    |
    +resources/
    |         |
    |         +alarmon.gif
    |         |
    |         +alarmoff.gif
    |
    +net/
        |
        +ponder2/
                |
                +managedobjects/
                                |
                                +AlarmClock.java
```

Your Java code will go in the `net/ponder2/managedobjects` package directory. Other files such as the alarm clock

GIF images and PonderTalk .p2 files will go in the `resources` directory.

Implementation

Now that we have the correct directory setup, we can create our Java code. Create a file in `managedobjects` called AlarmClock.java. We can start with the following code. Since we are creating a Swing window we need to extend JFrame. Also since we are creating a managed object we need to implement the `net.ponder2.ManagedObject` interface. There are no required methods for this interface, it is solely required to tell the Java compiler to start creating stub code to map the PonderTalk calls into the class.

Note: Turn off line numbers if you want to copy and paste the code

Toggle line numbers

```
package net.ponder2.managedobject;

import java.awt.*;
import javax.swing.*;
import net.ponder2.*;
import net.ponder2.apt.Ponder2op;

class AlarmClock extends JFrame implements ManagedObject {

}
```

Now we need to populate the class with constructor(s) and method(s). First, we need one constructor per factory message. In this case we only have the `create:` message so only need one constructor:

Toggle line numbers

```
    // The GIF images are stored here
    private ImageIcon alarmon, alarmoff;

    // This label is given the appropriate GIF image at the right time
    private JLabel label;

    @Ponder2op("create:")
    public AlarmClock(String aString) {
        // Set the window title to the PonderTalk supplied string
        setTitle(aString);

        // We need some standard Swing setup code


        // Create icons for the alarm and a label to hold it
        alarmon = new ImageIcon(this.getClass().getResource("/resource/alarm_clock-on.gif"));
        alarmoff = new ImageIcon(this.getClass().getResource("/resource/alarm_clock-off.gif"));
        label = new JLabel();
        label.setIcon(alarmoff);

        // Add the image to the frame's content pane;
        getContentPane().setLayout(new BorderLayout());
        getContentPane().add(label, BorderLayout.CENTER);

        // Adjust the frame size to macth the GIF images
        int width = alarmoff.getIconWidth();
        int height = alarmoff.getIconHeight();
        setSize(width, height+30);

    }
```

The **Ponder2op** annotation tells the compiler to generate stub code to map the PonderTalk message `create:` to this constructor. Note the use of the ":" character, it tells the compiler that there will be one argument that is to be passed from the PonderTalk command to the constructor. More information about the annotation can be found at

Ponder2opAnnotation.

Now we need to add methods for the operational commands, one per command. We have `visible:`, `visible`, `alarm:`, `alarm`. We can use the Ponder2op annotation is a similar manner:

Toggle line numbers

```
    private boolean visible = false;

    @Ponder2op("visible:")
    public void setAlarmVisible(boolean aBoolean) {
        // Call the JFrame method
        setVisible(aBoolean);
    }

    @Ponder2op("visible")
    public boolean isAlarmVisible() {
        // Call the JFrame method
        return isVisible();
    }

    @Ponder2op("alarm:")
    public void setAlarm(boolean aBoolean) {
        label.setIcon(aBoolean ? alarmon : alarmoff);
    }

    @Ponder2op("alarm")
    public boolean isAlarmSet() {
        return label.getIcon() == alarmon;
    }
```

Note that the method names do not have to match the Ponder2op annotations but it does make it clearer if they do. We now have a Java class which is also a Ponder2 managed Object.

Compiling

To compile our new class you can use the Ant `build.xml` file in the Ponder2 root directory. Simply enter the command:

```
ant build
```

Resources

Now we need the alarm clock images. These will go in the `resource` directory. They can be found in this zip file alarm_clock.zip. Download this file and extract it in the resource directory, it will create two files: `alarm_clock-on.gif` and `alarm_clock-off.gif`.

Running

We will see how to test the new managed object using the shell later. For the moment we will test it by writing some PonderTalk.

```
factory := root load: "AlarmClock".
alarm := factory create: "Wake Up".
// Put the alarm into the domain hierarchy so that the shell can access it later
root at: "alarm" put: alarm.
// Now show the alarm
alarm visible: true.
```

Save this file as `alarm.p2` in the `resource` directory. This may be run with the command:

```
ant run -Dboot=alarm.p2
```

If all goes well (and why not?) you should see a small alarm clock appear on your screen and it will just sit there. The

Ponder2 application is still running. We can now use the shell to test our alarm clock further:

Interacting with AlarmClock

We can use the Command Shell to test our new managed object. Assuming the above section was successful and the alarm window is on your screen, try the following commands

Step	*Command*	*Description*
1	`$ telnet localhost 13570`	Start a telnet session connected to the Ponder2 internal Command Shell
2	`$ ls`	List the root directory. You will see `alarm` there, that is your actual running Managed Object
3	`$ a := root/alarm`	Create a reference to the alarm managed object, it will save us typing `root/alarm` again
4	`$ a visible: false.`	Send the `visible:` message to the alarm, this will call the `setAlarmVisible(boolean)` method and hide the alarm
5	`$ a visible: true.`	Show the alarm again
6	`$ a alarm: true.`	Set the alarm ringing. You should see an animated clock in the window now
7	`$ a alarm.`	Ask for the status of the alarm

Use ctrl-C to kill the running Ponder2 process.

ManagedObjectsSimple (last edited 2009-11-18 17:18:31 by KevinTwidle)

ManagedObjectsAdvanced

Advanced Concepts

After writing your own simple Managed Objects you may find that you want to do more advanced operations like sending messages to other Managed Objects, receiving and understanding any message etc. This page contains short guides to help you achieve what you want to do.

Java Data Types

There are some basic Java classes that are important to know about when writing Managed Objects

P2Object

P2Object is the Java class that is the base class for all the Managed Objects in the Ponder2 environment. Whenever a Managed Object is provided in a PonderTalk message it arrives as an argument of type P2Object.

P2Objects are used extensively throughout the Ponder2 system to send messages, all PonderTalk message arguments are P2Objects. The P2Object class contains many static helper methods to wrap arguments up as P2Objects. e.g.

Toggle line numbers

```
P2Object string = P2Object.create("A string");

P2Object integer = P2Object.create(23);

P2Object float = P2Object.create(23.5);
```

P2ObjectAdaptor

A P2ObjectAdaptor is the base class for the stub code produced by the Java annotation compiler for translating PonderTalk messages into Java methods. The stub class for any Managed Object is <ManagedObjectClassName>P2Adaptor. So, for the simple Managed Object in the writing guide, the AlarmClock class has an adaptor class created at compile time called AlarmClockP2Adaptor. See the ObjectAdaptor page for more information about how this works.

@Ponder2op

While not strictly a Java data type, it is an important part of writing the Java Managed Object. The @Ponder2op annotation is used to tie a PonderTalk message to a Java method. The annotation is placed immediately in front of the method. It takes a single string as its argument being the PonderTalk message name. Note there is no semi-colon allowed after the Ponder2op annotation. e.g.

Toggle line numbers

```
@Ponder2op("at:put:")
public void add(String aName, P2Object aManagedObject) { ... }

@Ponder2op("+")
public void add(String value) { ... }
```

In the above example, the PonderTalk message name is `at:put:` which is a keyword message that takes two arguments (the colon characters are very important here). The compiler matches the two keyword arguments to the two method arguments and writes code into the stub adaptor class that maps the `at:put:` call to a call to the (in this case) *add* method and it converts the PonderTalk arguments to the correct types e.g. String and P2Object.

The second case is an example of a binary message with therefore takes one and only one arguement.

Thus the following PonderTalk will be handled correctly:

Toggle line numbers

```
1 myobj at: "Fred" put: root/mydomain/anotherobject.
2
3 myobj + "Some string".
```

In fact this would also work should a number be used for the `at:` argument rather than a string because the number would be transparently converted to a string and then passed to the method. Should any argument not be able to be converted to the expected type an error will be thrown before the method can be called. This means that the Java Managed Object does not have to deal with errors of this kind which makes the programming easier.

Ponder2Exception

This class acts as a base class for three exception classes: Ponder2ArgumentException, Ponder2OperationException and Ponder2RemoteException. None of these exceptions need be caught by a Java Managed Object by a managed object's methods may throw one or more of them as appropriate. They are dealt with by the overall Ponder2 system.

Ponder2ArgumentException

This exception is thrown if there is something wrong with the arguments. Typically it is to o with the conversion of the PonderTalk arguments to the types required by the method. Efforts are made by the system to convert the arguments if possible but sometimes this fails. E.g. if an integer is expected and a string is given then if it can be converted to a number it is, otherwise a Ponder2ArgumentException is thrown. After this exception is thrown, the current PonderTalk filename, line number and character number are added to the exception.

Ponder2OperationException

This exception is thrown when something has gone wrong in a method. The argument string should include details of the error. After this exception is thrown, the current PonderTalk filename, line number and character number are added to the exception.

Ponder2RemoteException

This exception is thrown in the event of underlying communication failures. It is not produced by Managed Objects and is only included here for completeness.

Sending messages to other Managed Objects

PonderTalk messages are sent to Managed Objects using the `operation` method call. It is defined as:

Toggle line numbers

```
1 P2Object operation(P2Object source, String operation, P2Object... args) throws
Ponder2Exception;
```

Where `source` is the object that sends or initiates the message, operation is the PonderTalk message identifier e.g. "at:put:" and args is an array of zero or more P2Objects. The source of the message is important for authorisation purposes.

There are two distinct cases where you need to send a message to another managed object, one is where you initiate a message as the result of an external action or event e.g. someone interacting with a GUI, a timer has expired, a file is changed, a TCP message is received from a non-Ponder2 source etc. The other is where your Managed Object has received a message from another Managed Object and as a result sends a message on elsewhere.

In the first case where the message is initiated by your Managed Object, you must include your P2Object identifier as the source of the operation. In the second case, you need to acquire the source of the message you have received and use that as the source of the operation you perform (unless you want to use *your* object's authorisation permissions for the operation).

As seen in the example of a simple Managed Object, no mention is made of P2Objects to keep things easy. You can write code that responds to PonderTalk messages without having to know anything about the inner workings of the Ponder2 environment. Since your Java class only has to implement an interface and can extend any class it likes, the relevant P2Objects are not normally available.

Initiating a message to another Managed Object

Your object's adaptor class containing all the PonderTalk message name to method mappings is actually the P2Object representing the instance of your Managed Object. This is because P2Object is a base class of your adaptor stub code. If you initiate an operation on another Managed Object you must use your P2Object identifier. To get your P2Object identifier you can simply include an extra argument in your object's contructor(s). Instead of something like:

Toggle line numbers

```
@Ponder2op("create")
MyObject() {
    this.title = "An Object";
}

@Ponder2op("create:")
MyObject(String aTitle) {
    this.title = aTitle;
}
```

You need to add an extra argument called `myP2Object` like this

Toggle line numbers

```
@Ponder2op("create")
MyObject(P2Object myP2Object) {
    this.myP2Object = myP2Object;
    this.title = "An Object";
}

@Ponder2op("create:")
MyObject(P2Object myP2Object, String aTitle) {
    this.myP2Object = myP2Object;
    this.title = aTitle;
}
```

The extra argument is recognised by the compiler (it must be called `myP2Object` and be of type `P2Object`) and it is filled in by the system when the object is created. It can be saved as an instance variable and used as necessary when sending messages. It is not part of the normal PonderTalk message arguments and so is not represented in the Ponder2op annotation.

Sending a message to another Managed Object as a result of receiving a message

When a message is received from another Managed Object it arrives in the form of a method call with the appropriate arguments. To get the P2Object identifier of the sender you can simply include an extra argument in your method. Instead of something like:

Toggle line numbers

```
// Set the GUI so that we can later display our data
// PonderTalk: obj gui: root/guis/agui.
@Ponder2op("gui:")
public void setGui(P2Object guiobj) {
    this.gui = guiobj;
}

// Send the data to the GUI window
@Ponder2op("dumpData")
public void() {
    // We want to send the GUI a message but don't have the sender's P2Object id
}

// An example to show more than one argument
@Ponder2op("at:put:")
public void store(String name, String value) {
    myStore.add(name, value);
```

```
}
```

You need to add an extra argument called `source` like this

Toggle line numbers

```
// Set the GUI so that we can later display our data
// PonderTalk: obj gui: root/guis/agui.
@Ponder2op("gui:")
public void setGui(P2Object guiobj) {
    this.gui = guiobj;
}

// Send the data to the GUI window
// operation() may throw an error, it should be passed up
@Ponder2op("dumpData")
public void(P2Object source) throws Ponder2Exception {
    gui.operation(source, "display:", P2Object.create("Some data"));
}

// An example to show more than one argument
@Ponder2op("at:put:")
public void store(P2Obect source, String name, String value) {
    myStore.add(name, value);
    // We could use source for something here
}
```

The extra argument is recognised by the compiler (it must be the first argument and called `source` and be of type `P2Object`) and it is filled in by the system when the method is called. It can be used to "pass" the message on to another object, in which case the message source's authorisation will be used for the message. This argument is not part of the normal PonderTalk message arguments and so is not represented in the Ponder2op annotation.

Wildcard messages

There may be the case where you want to write a Managed Object that can accept any message, interpret it and perform actions based on that message. For this case the wildcard Ponder2op may be used with a method that takes a particular set of arguments. The method name may be anything but the argument prototype should be the same as:

Toggle line numbers

```
// Receive all messages here
@Ponder2op(Ponder2op.WILDCARD)
    protected P2Object message(P2Object source, String op, P2Object... args) {
    P2Object result;
    ...
    return result;
}
```

The op string contains the PonderTalk message identifier which is normally used in the Ponder2op annotations e.g. `at:put:`, `display`, `+` etc. etc.

ManagedObjectsAdvanced (last edited 2008-09-10 08:13:02 by KevinTwidle)

Ponder2 Policies

Policies are rules governing choices in the behaviour of the system. There are two basic types supported by the current implementation: Obligation Policies and Authorisation Policies.

- Obligation Policies, also known as Event Condition Action rules must perform certain actions when certain events occur.
- Authorisation Policies permit or deny actions based upon the action, the source of the action and the target of the action.

Policies are dynamic, they may be changed at any time by loading, enabling and disabling without stopping the system. Policies may be specified for groups of managed Objects, often before the objects themselves are instantiated. Full documentation about the different types of policies may be found by following the links above.

Ponder2Policies (last edited 2008-01-25 16:12:58 by KevinTwidle)

Obligation Policies

Contents

Policies are the Event Condition Action rules of the Ponder2 system. An obligation policy is a Managed Object that is instantiated and given the event it should be expecting, zero or more conditions to be evaluated and one or more actions to be performed if the conditions are satisfied.

Obligation policies are created from the Event, Condition, Action policy factory:

Toggle line numbers

```
1 newPolicy := root/factory/ecapolicy create.
```

Event

Once created, the policy must be given the information it needs to perform its function. We have to give the policy the Event Template we want it to respond to and the conditions and actions it must use when an event of that type is generated. Here an event template is created first to be used in the following examples.

Toggle line numbers

```
1 template := root/factory/event create: #( "monitor" "value" ).
2 root/event at: "monitor" put: template.
```

This event template may be used by a monitoring system to generate events. Each event will have the monitor name and the value from that monitor as attribute values. Once the event template managed object has been created it can be given to the policy as the expected event type.

Toggle line numbers

```
1 policy := root/factory/ecapolicy create.
2 policy event: root/event/monitor.
```

Condition

Policies may have zero or more conditions. If it is desired that the policy's actions are to be executed every time hat event occurs then no condition need be specified otherwise conditions are required. Each condition is simply a PonderTalk block that will be executed when the event occurs. The PonderTalk block must return a boolean. If there is more than one condition then all the conditions are executed and the logical AND of all the conditions is taken to be the final result. If the result is true then the action(s) is/are performed. NB. There is no guaranteed order to the execution of multiple conditions, nor whether they are run concurrently nor whether they are all evaluated if one condition returns false. If you are worried about any of these issues then use one more complex condition.

Toggle line numbers

```
1 policy condition: [ :value | value > 100 ].
```

Action

Toggle line numbers

```
1 policy action: [ :monitor :value | root print: "Monitor " + monitor + " has value " +
value ].
```

Toggle line numbers

```
1 policy active: true.
```

Example

This probably looks a little better laid out using cascading messages:

Toggle line numbers

```
1 policy := root/factory/ecapolicy create.
2 policy event: root/event/monitor;
3        condition: [ :value | value > 100 ];
4        action: [ :monitor :value |
5             root print: "Monitor " + monitor + " has value " + value
6        ];
7        active: true.
```

Anatomy of a Policy

A policy, called */policy/testpolicy*, that uses *testevent* can be described as follows:

XSLT option disabled, please look at HelpOnConfiguration.

```
<use name="/policy">
  <add name="testpolicy">
    <use name="/template/policy">
      <create type="obligation" event="/event/testevent" active="true">
        <arg name="colour"/>
        <arg name="intensity"/>
        <condition>
          <and>
            <eq>!colour;<!-- -->red</eq>
            <gt>!intensity;<!-- -->34</gt>
          </and>
        </condition>
        <action>
          <!-- Add one to a counter managed object -->
          <use name="/dom1/counter">
            <inc/>
          </use>
        </action>
      </create>
    </use>
  </add>
</use>
```

Breakdown of Policy XML

Create a new Policy called /Policy/testpolicy

```
<use name="/policy">
  <add name="testpolicy">
    <use name="/template/policy">
      <create ...>
        ...
      </create>
    </use>
  </add>
</use>
```

The policy will be an ECA type policy ("obligation") and it will respond to events of type */event/testevent*, described here. The policy will become active as soon as it is created.

```
<create type="obligation" event="/event/testevent" active="true">
  ...
</create>
```

The policy will make use of two named arguments that the event provides:

```
<create event="/event/testevent" ...>
  <arg name="colour"/>
  <arg name="intensity"/>
  ...
</create>
```

Named arguments are substituted as text before the XML action is evaluated by enclosing the name inside the two characters "!" and ";"

Condition

The optional condition can contain simple boolean statements comparing string and integer values. In this case we are checking whether the colour is *red* and the intensity is greater than *34*. Conditions can contain any combination of *and*, *or*, *not*, *eq*, *ne*, *gt*, *ge*, *lt* or *le*. *And* and *or* take any number of XML sub-elements, *not* takes one, the others all take two. Note the string substitution of the arguments *colour* and *intensity*. Note also that the arguments for the comparisons have been separated by XML comments to ensure that they are separate XML elements.

```
  <condition>
    <and>
      <eq>!colour;<!-- -->red</eq>
      <gt>!intensity;<!-- -->34</gt>
    </and>
  </condition>
```

Action

The action part of the ECA policy is mandatory. It simply consists of Ponder2 XML, the only difference being that execution of the XML is delayed until the policy's event and condition parts are satisfied.

```
 <action>
    <!-- Add one to a counter managed object -->
    <use name="/dom1/counter">
      <inc/>
    </use>
  </action>
```

CategoryPonder2Project CategoryUsingPonder2

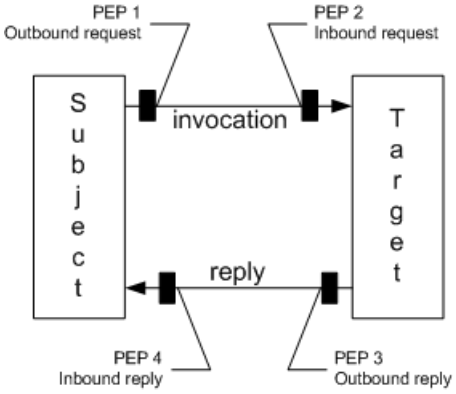
ObligationPolicies (last edited 2008-01-25 12:39:01 by KevinTwidle)

Ponder2Authorisation

Authorisation Framework in Ponder2

Contents

Ponder2 supports Authorisation Policies to control interactions between managed objects. The Ponder2 Authorisation Framework (PAF) introduces novel ideas on the granularity of control of authorization policies. In particular, in PAF authorisation policies can be uniformly specified and enforced for protecting both the subject and the target for a given action.



As the Figure above shows, PAF provides 4 policy enforcement points (PEP):

- PEP1 and PEP4 are used to enforce authorisation policies for the subject side
- PEP2 and PEP3 are used to enforce authorisation policies for the target side

By enforcing policies at PEP1, it becomes possible to specify authorisation policies that prevent subjects from performing actions that could be harmful for them or their domain(s), e.g. preventing a web browser sending a request to a blacklist webserver. Furthermore, enforcement of policies at PEP4 could prevent a subject from accepting a reply from an action that could threaten the integrity of the subject.

For the target side, PEP2 can be used to enforce traditional access control authorisation policies. Additionally, when authorisation policies are enforced at PEP3, it becomes possible to protect the privacy of the target that could be compromised when the result of an action contains information that should not be revealed (e.g., by applying an authorisation policy that filters out the sensitive data from the result).

PAF supports both negative and positive authorisation policies. When two or more policies of opposite sign apply to the same action *modal conflicts* may be introduced. For this reason, PAF provides a conflict resolution strategy (based on domain nesting precedence) that deals with such conflicts at runtime.

In the following, we will provide a more detailed description of the PAF and its capabilities by using several scenarios. Each scenario consists of a set of managed objects together with several authorisation policies to control the execution of the managed objects' actions. The scenarios can be downloaded and executed.

HospitalDomain contains a description of the case study that is used in all the scenarios.

BasicScenario introduces how to define basic authorisation policies.

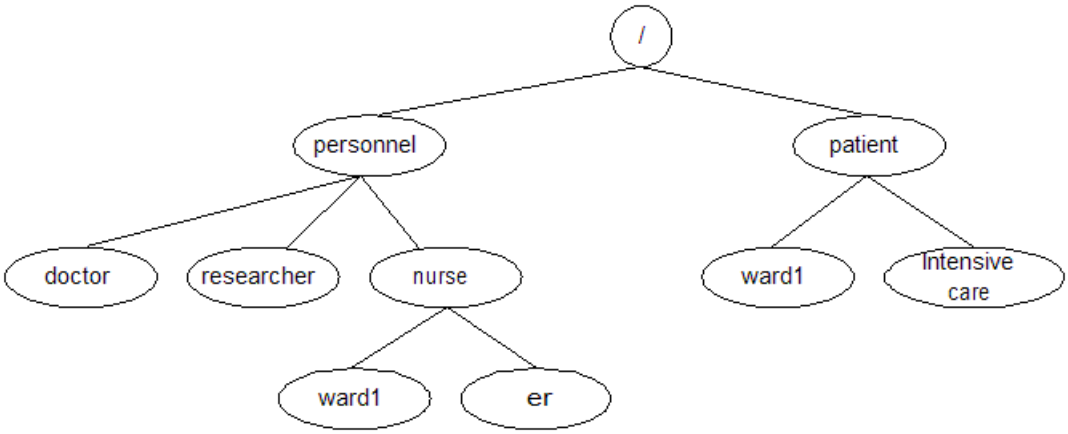
AdvancedScenarios describes how to define more advanced policies and provides some details on the conflict resolution strategy that PAF uses.

Publications on this work are available in Ponder2 Publications

HospitalDomain

The Hospital Domain Structure

All the scenarios described in the following use the same domain structure that is shown in the Figure below. This structure represents a Hospital organization where carers and hospital personnel are contained in the the *personnel* subdomain and patients are organised in the *patient* subdomain. Personnel is further organised according to specialisation and to the location where on duty (such as a ward and /or an emergency room).



In the scenarios that will follow we make use of three managed objects, *Patient*, *Nurse*, and *Researcher*.

The *Patient* object is used to represent patient information. It has three fields, *name, age* and *symptom*, and it provides the following two methods to retrieve a patient's information:

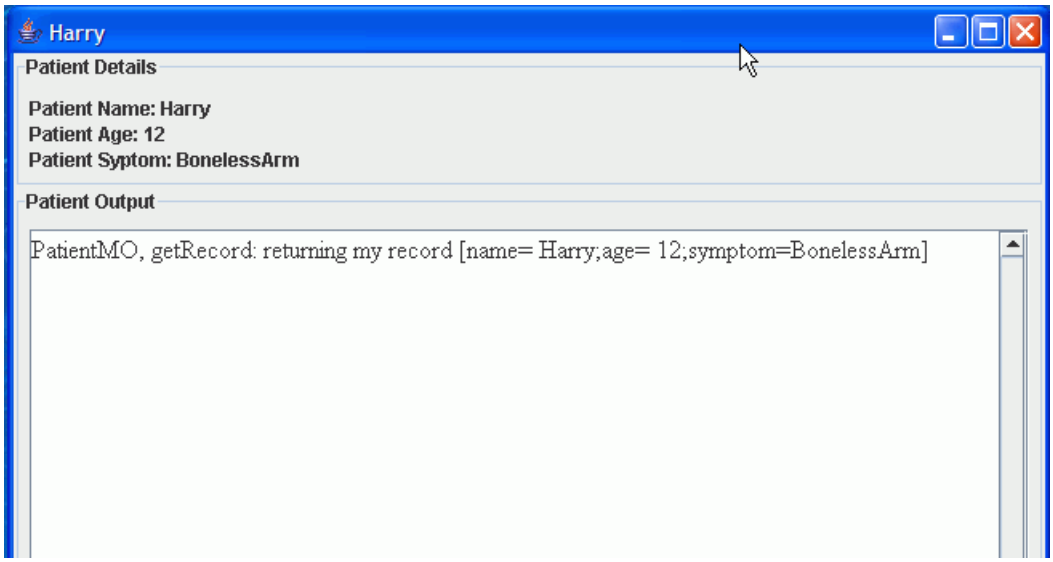
```
getRecord(nurselevel)
```

that requires the nurse level and returns a string containing all the information regarding a patient. This string is constructed by a concatenation of all the patient fields (name+age+symptom).

```
getData()
```

instead only returns a string that contains the age and the symptom of the patient.

The patient managed object also provides a GUI shown below:



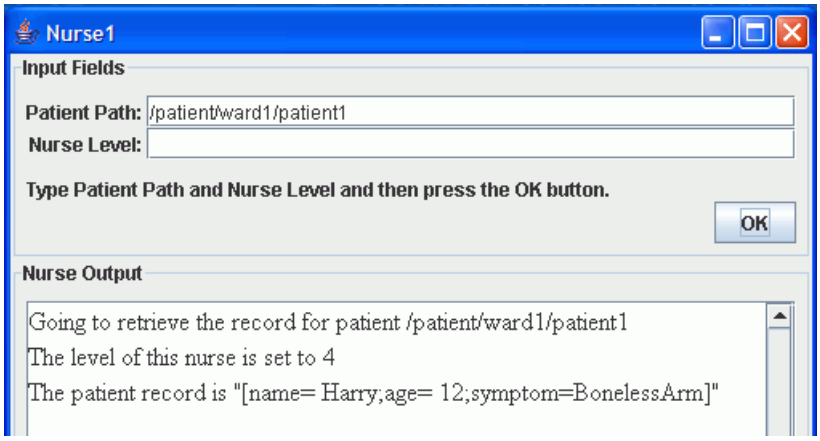
The Patient Details panel shows the details of a the patient. The Patient Output panel provides a text area that outputs information of each executed operation. For instance, in the case of the Figure above, the text area says that a *getRecord* operation is invoked. The content of the returned record is provided as well.

The *Nurse* managed objects has a field *nurseLevel* representing the nurse experience and a method

```
checkRecord(patientPath, nurseLevel)
```

that retrieves the record of a patient in the *patientPath* invoking the patient's method *getRecord()*. The *patientPath* specifies the path from the root domain to the patient managed object.

As for the patient, the nurse is also provided of a GUI that is shown below:



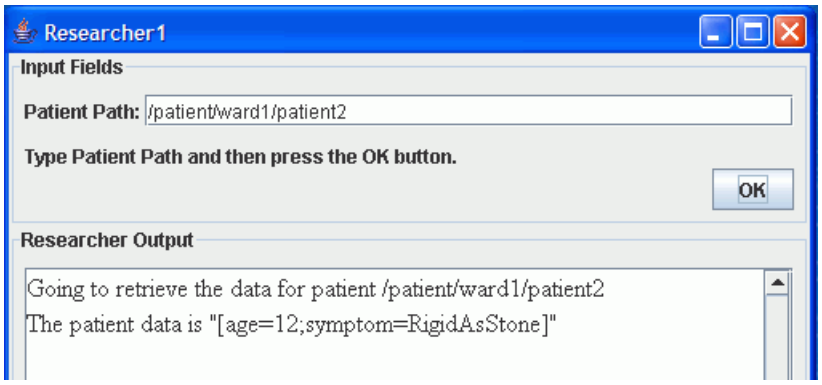
The Nurse GUI provides an Input Fields panel, where the patient path and the nurse level can be inserted by a user. The nurse level value can be omitted during an invocation. If this is the case, the default level is used (that is 4, as shown in the figure above). The Nurse Output panel shows information about the operation that is performed, such as the path of the patient managed object that is going to be invoked and the level of the nurse together with the result of operation. If the operation succeeds, the value of the record is shown. Otherwise, information is provided about the cause of the operation failure.

Finally, the *Researcher* managed object has the following method:

```
checkData(patientPath)
```

that invokes the patient's method *getData()* of a given patient specified by *patientPath*.

The researcher managed object provides a GUI and a screenshot is shown below:



The Input Fields panel allows a user to specify the patient from which retrieving the data. The Output panel outputs the operation being executed. In this instance, the researcher got the data of patient2 in ward1.

HospitalDomain (last edited 2008-05-17 20:43:01 by GiovanniRussello)

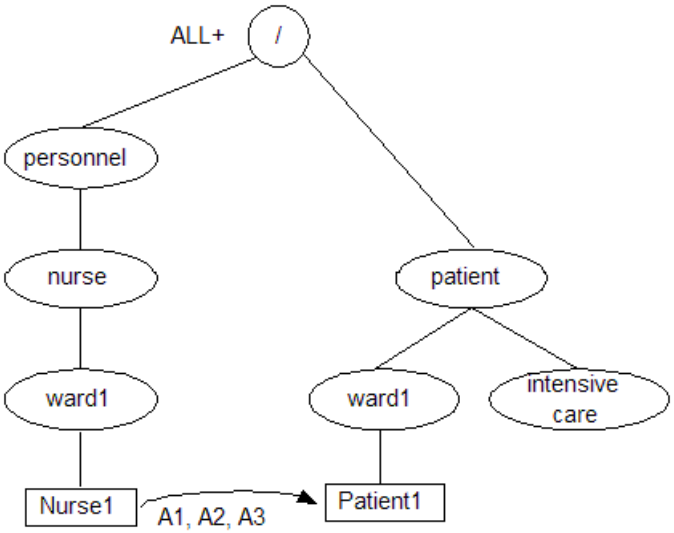
BasicScenario

Getting Started with the Basic Scenario

For getting started in the use of authorization policies let us consider a simple scenario and some examples of authorization policies. Let us assume that the default authorization policy is ALL+, meaning that all actions are authorized unless a negative authorization policy is specified. Let us start the Ponder2 interpreter loading the files for building the hospital domain structure and the managed objects. The command line arguments are shown below:

```
-auth allow -boot hospital/hdomain.p2 -boot hospital/basic/nurse-settings.p2 -boot
hospital/basic/patient-settings.p2
```

The option ***-auth allow*** tells the interpreter to instantiate the PAF and that the default authorisation policy is ALL+.The file *hospital/hdomain.p2* contains the definitions for loading the hospital domains as shown in the Figure below. The file *hospital/basic/nurse-settings.p2* imports a *Nurse* managed object in the /personnel/nurse/ward1 path. Similarly, the file *hospital/basic/patient-settings.p2* loads a *Patient* managed object in the /patient/ward1 path.



The setting file for the patient contains also the definitions of the authorisation policies A1, A2, and A3.

The code excerption below defines the authorisation policy A1.

Toggle line numbers

```
21 root/tauthdom at: "a1" put: (newauthpol subject: root/personnel/nurse/ward1/nurse1
22                                         action: "getrecord"
23                                         target: root/patient/ward1/patient1
24                                         focus:"t").
25 root/tauthdom/a1 reqneg.
26 root/tauthdom/a1 active: false.
```

In line 21, the new policy is created in the domain /tauthdom (target authorisation domain) using the factory ***newauthpol***. When an authorisation policy is created 4 arguments need to be given. The first argument sets the path of the *subject* of the action. In this case, the subject path points to nurse1. The second argument specifies the action that is controlled by this authorisation policy, that in this case is the *getrecord* method (see line 22). The third argument is the path to the *target*. The last argument specifies the focus of the policy. As we said, our authorisation policies can be specifies for either a subject or a target side of an action. If this argument is set to "s" then the policy is a subject authorisation, meaning that it will be enforced at PEP1 and PEP4. While setting the argument to "t" (as in line 24) the authorisation policy will be enforced at PEP2 and PEP3.

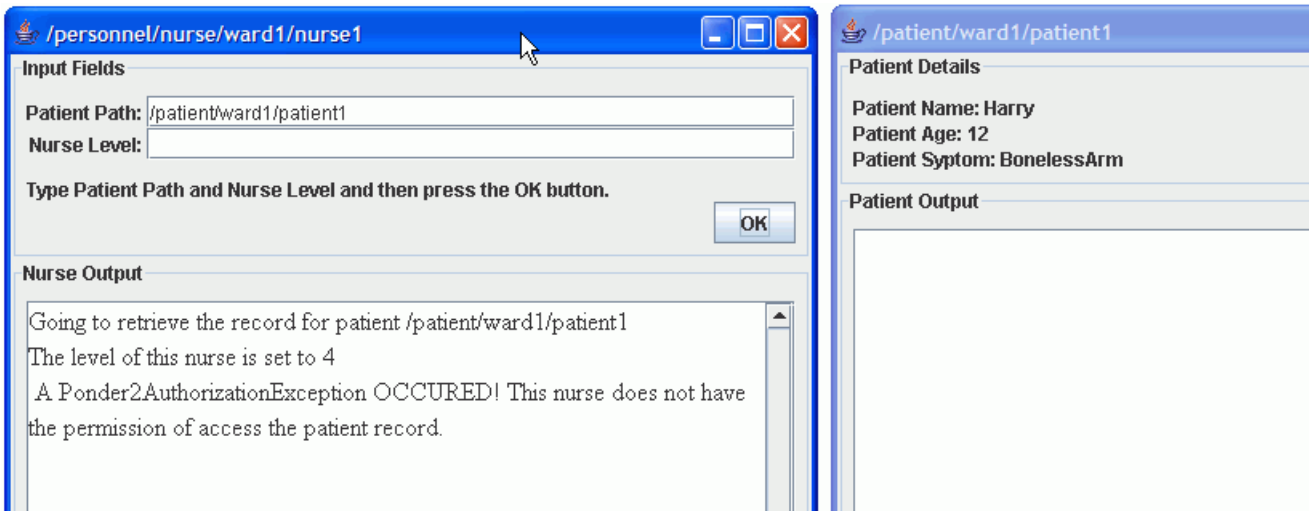
Line 25 sets that the policy is negative for the incoming part. This means that if such policy is enforced at PEP2 then the nurse1 cannot invoke the getRecord method of patient1. Line 26 is used to (de)activate the policy. As it is, the policy is not active and will not have any effect on the execution of the action. If we run the interpreter with this setting and invoke the getRecord method from nurse1 then the operation will succeed.

Now, let's activate policy A1 setting active to true, as show in line 26 below:

Toggle line numbers

```
26 root/tauthdom/a1 active: true.
```

Starting again the interpreter and executing the action from the nurse GUI we obtain the following result:



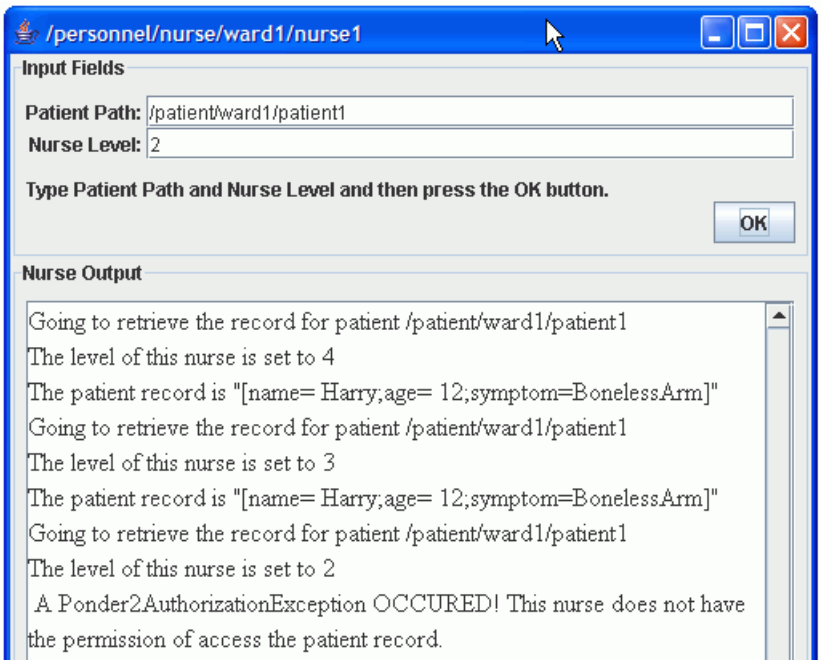
As we can see from the output panels of the nurse's GUI, the nurse managed object tried to access the record of patient1 but an authorisation exception occurred. This because at PEP2 the authorisation policy A1 was enforced and since A1 is a negative authorisation policy the action was not allowed.

Next, let's deactivate policy A1 and activate policy A2 that is shown below:

Toggle line numbers

```
33 root/tauthdom at: "a2" put: (newauthpol subject: root/personnel/nurse/ward1/nurse1
34                                         action: "getrecord"
35                                         target: root/patient/ward1/patient1 focus:"t"
).
36 root/tauthdom/a2 reqneg.
37 root/tauthdom/a2 reqcondition: [ :nurselevel | nurselevel < 3]. //set the condition
38 root/tauthdom/a2 active: true.
```

This policy is similar to A1 except that it as a condition on the incoming part. Conditions can be specified on the arguments of the method that the policy controls. In particular, the condition in policy A2 is specified on `nurselevel` that is an argument of the getrecord method. This means that policy A2 is applicable only to nurses with a level less then 3. In other words, when policy A2 is enforced, any nurses that have a level less then 3 are not allowed to access patient1's record. Basically, if we run the example with the default nurse level (which is 4) then the action is executed. When the level is set to 2 then an authorisation exception occurs. The figure below shows the nurse GUI after the execution of the operation using the default nurse level, a level of 3 and finally a level of 2. When the level was set to 2 the authorisation exception is thrown:



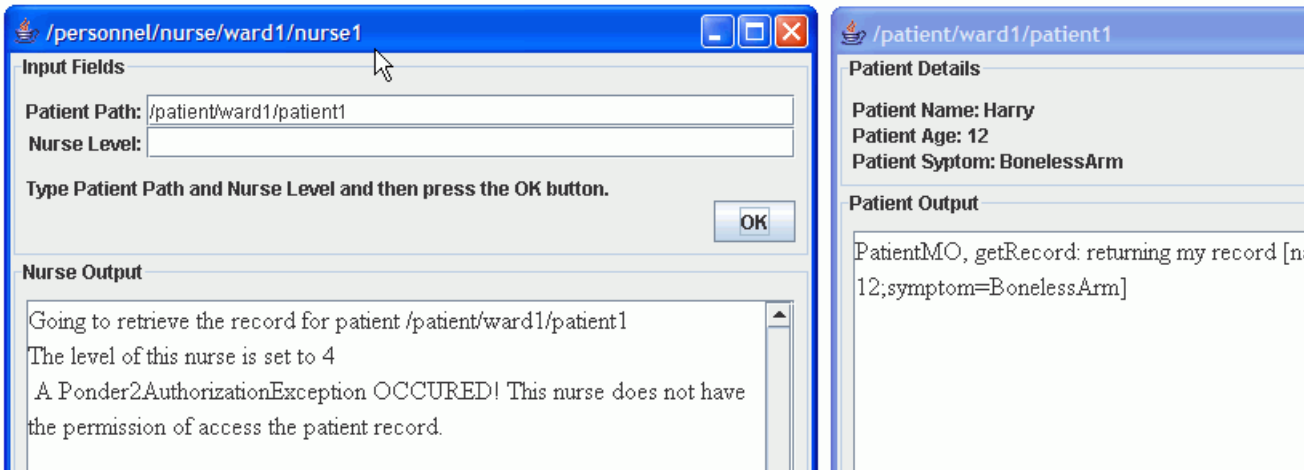
This behaviour can be explained as follows. The only authorisation policy that is active is policy A2. When the PAF search for a valid policy to enforce at PEP2, A2 is selected. However, before applying the policy its condition must be evaluated. For the first two executions the condition yelds false (since the level of the nurse is never strictly less than 3) and the policy is not applicable. Since there are no other policies, the PAF uses the default authorisation policy which is ALL+ and the action is allowed. In the last execution however, the condition yelds true, the policy A2 is enforced and the action is not allowed.

Finally, the last policy to consider is policy A3 that is shown in the excerpt below

Toggle line numbers

```
45 root/tauthdom at: "a3" put: (newauthpol subject: root/personnel/nurse/ward1/nurse1
46                                         action: "getrecord"
47                                         target: root/patient/ward1/patient1 focus:"t"
).
48 root/tauthdom/a3 reqneg.
49 root/tauthdom/a3 reqcondition: [ :nurselevel | nurselevel < 3].
50 root/tauthdom/a3 repneg.
51 root/tauthdom/a3 repcondition: [ :patrecord | patrecord =="[name= Harry;age=
12;symptom=BonelessArm]"].
52 root/tauthdom/a3 active: true.
```

Similarly to A2, policy A3 has a condition for the requesting part (PEP2). Additionally, A3 is set to be a negative authorisation policy also for the replying part (PEP3) with a condition block, as specified in line 50 in line 51. respectively. The condition in line 51 says that if the patient's record (that is the value returned by the action) is equals to "[name= Harry;age= 12;symptom=BonelessArm]" then the result is not to be returned. The execution of the scenario when such a policy is activated is shown in the figure below:



At PEP2 policy A3 is not enforced since the nurse level was greater then 3 and its condition for the incoming part yelds false. The action is executed by the patient1 (as we can see from the patient GUI in the right hand side). However, at PEP3 the policy A3 is enforced since the return condition yelds true and an authorisation exception is thrown.

The AdvancedScenarios page describes how to define more advanced policies and provides some details on the conflict resolution strategy that PAF uses.

BasicScenario (last edited 2008-05-17 20:53:17 by GiovanniRussello)

AdvancedScenarios

Policy Conflicts and Resolution

When dealing with policy based systems, it is unavoidable that conflicts arise in the set of policies. Ideally conflicts are detected by static analysis of the policy set. However it is often not possible to perform such analysis on policies that depend on run-time state. In order to overcome such a issue, we designed a strategy that deterministically resolves the conflicts between two or more policies that apply to the same (subject, target, action)-triple.

To determine the precedence between two or more policies we based our conflict resolution algorithm on domain nesting. The domain nesting resolution gives precedence to policies that apply to a more specific instance of subjects, targets, or both. In other words, a policy that applies to a subdomain is more specific than a policy that applies to any ancestor domains. The main strength of this approach is that it is intuitively applicable to a domain-based system. However, it is possible that policies could be specified on subjects that are at different levels in the domain structure but on the same target, and vice-versa. In the following, we present several cases of such conflicts and we discuss how our strategy resolve them.

There are several case in which conflicts can arise and we provide an overview in the following scenarios:

Scenario1

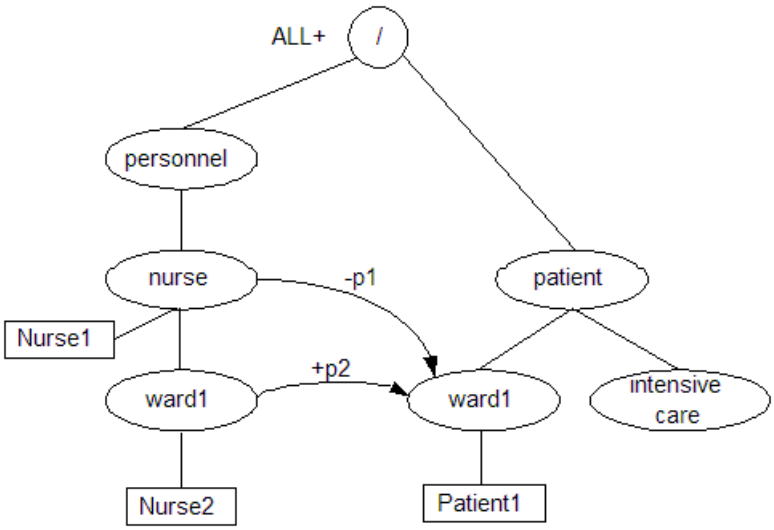
Scenario2

Scenario3

Scenario1

Scenario 1: policies converging to the same target domain

Let us consider the case in which two policies are specified as shown in the figure below:



This scenario represents the case when an action on a target in general is prohibited to a group of subject, while it is authorized for a restricted group of entities.For instance, in general, nurses are not allowed to access a patients' records (negative policy p1), but only if a nurse is on duty on the same ward of the patient (positive policy p2). This case results in a modal conflict between policies p1 and p2 when Nurse2 invokes the getRecord() method of Patient1. The PAF can resolve automatically this type of conflict giving higher priority to the most specific policy. In this case, policy p2 would get enforced.

To run this scenario the following command line has to be used:

```
-auth allow  -boot hospital/hdomain.p2 -boot hospital/scenario1/nurse-settings.p2 -boot
hospital/scenario1/patient-settings.p2
```

The policies p1 and p2 are defined in file patient-settings.p2 as follows:

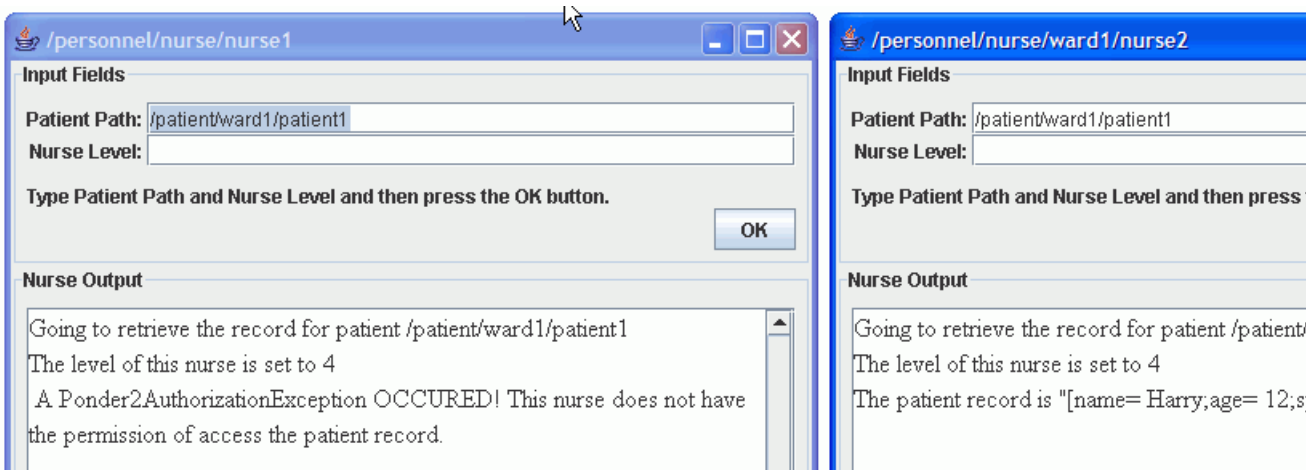
Toggle line numbers

```
root/tauthdom at: "p1" put: (newauthpol subject: root/personnel/nurse
                                        action: "getrecord"
                                        target: root/patient/ward1 focus:"t" ).
root/tauthdom/p1 reqneg.
root/tauthdom/p1 active: true.
```

Policies can be have both as subject and target domains. For instance, policy p1 has as subject the *nurse* domain and as target the *ward1* in *patient* domain. Policies defined on a domain *d1* are applicable to all instances of managed objects contained in *d1* as well as to any other managed objects contained in its subdomains. This means that policy p1 can be enforced for both Nurse1 and Nurse2. Any invocations from Nurse1 is bound to fail. However, for Nurse2 there isa positive policy p2 that takes priority and it is defined as follow:

```
root/tauthdom at: "p2" put: (newauthpol subject: root/personnel/nurse/ward1
                                        action: "getrecord"
                                        target: root/patient/ward1
                                        focus:"t" ).
root/tauthdom/p2 active: true.
```

Because policy p2 is more specific than policy p1, when Nurse2 makes the invocation policy p2 is enforced and the execution succeeds as shown in the screenshot below:



On the left-hand side, the Nurse1 execution causes an authorisation exception to be thrown, this because its call is not authorised by policy p1. On the other hand, the call from Nurse2 succeeds because of policy 2.

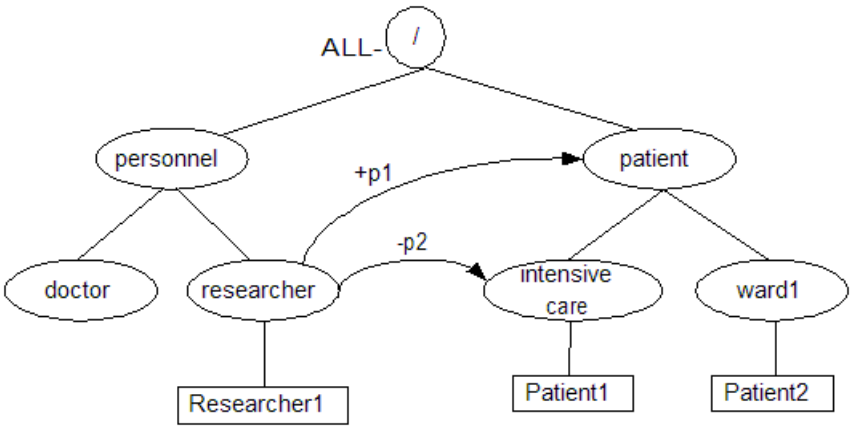
See also: Scenario2 and Scenario3

Scenario1 (last edited 2008-01-11 13:09:56 by KevinTwidle)

Scenario2

Scenario 2: policies originated from the same subject domain

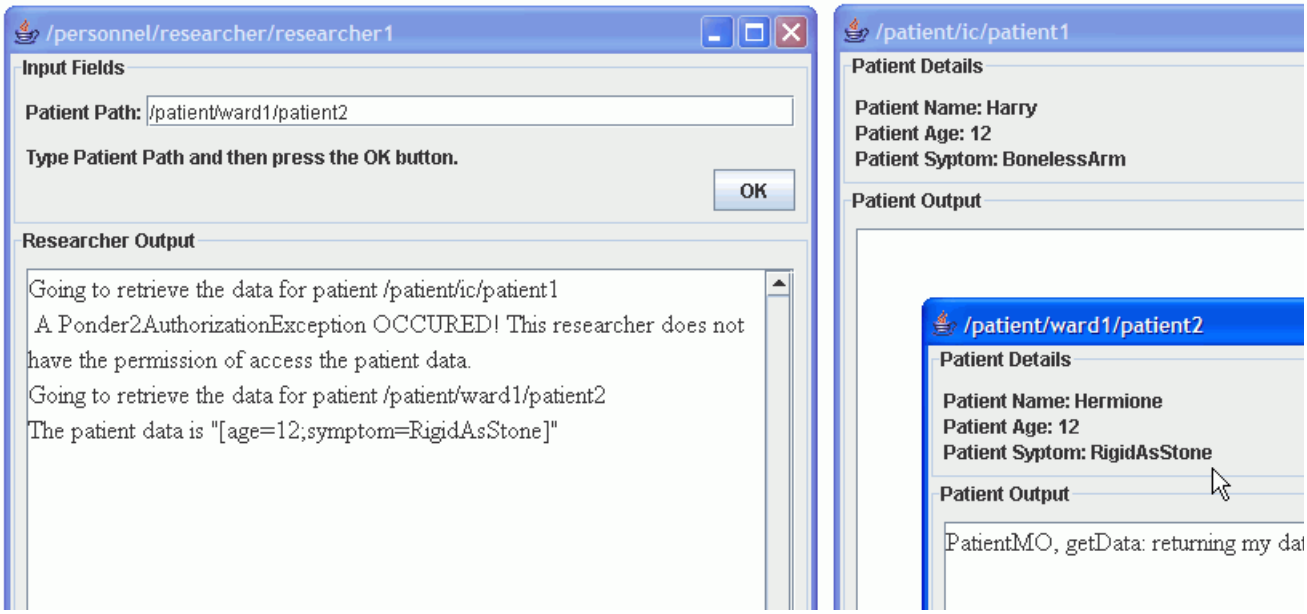
In this scenario, the execution of an action on a specific domain of entities must not be allowed. However the same action is permitted for a more general set of targets. The scenario is depicted in Figure 7, where medical researchers are allowed to access all the patients' data expect for the ones assigned to the intensive care ward. This leads to another conflict that it is resolved using the most specific of the two policies. The domain structure for this scenario is shown below:



The execution of this scenario is obtained by using the following command line:

```
-auth deny  -boot hospital/hdomain.p2 -boot hospital/scenario2/researcher-settings.p2 -boot
hospital/scenario2/patient-settings.p2
```

Note that to set the default policy value as ALL- we use the option *-auth* with value ***deny***. When the researcher GUI is pointed to Patient1 in ic (intensive care) the execution of the *getData* method is not authorized by policy p2. On the other hand, when the Patient2's path is used then the operation is authorized by policy p1 and the name of the patient is returned. The screenshot below offers a view of this execution:



We should point out that when the ALL- policy is used, then all actions in every PEP must be authorised. For instance, the researcher must have a subject authorisation policy for executing the *getData* action. This policy is defined in the *researcher-settings.p2* file as it is shown below:

Toggle line numbers

```
16 root/sauthdom at: "p1" put: (newauthpol subject: root/personnel/researcher
17                                         action: "getdata"
```

```
18                                          target: root/patient
19                                          focus:"s" ).
20 root/sauthdom/p1 active: true.
```

Changing the precedence rule

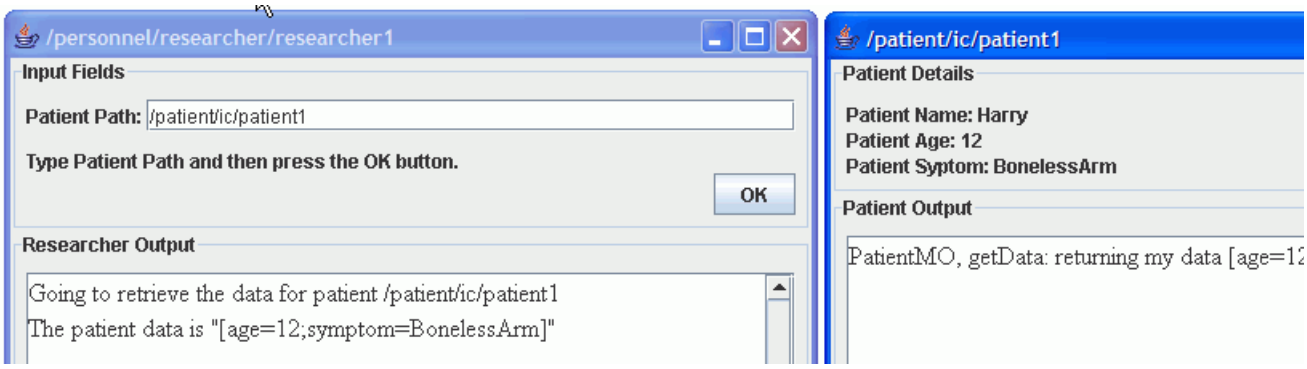
As we said, the PAF uses a conflict resolution strategy where precendence is given to most specific policies. However, this principle does not apply to all situations. Sometimes it is desirable that a global policy overrides more specific ones. For supporting these cases, in our framework it is possible to use special global policies that override any specific policies defined in the subdomain structure. To define such a policy the keyword ***final*** must be used in the definition of the policy. Final authorization policies can only be defined on domains, since it does not make sense to define such policies on object instances.

To make a concrete example, let us consider the scenario above and change the defition of policy p1 as following:

Toggle line numbers

```
21 root/tauthdom at: "p1" put: (newauthpol subject: root/personnel/researcher
22                                          action: "getdata"
23                                          target: root/patient
24                                          focus:"t").
25 root/tauthdom/p1 final.
26 root/tauthdom/p1 active: true.
```

In line 25 policy p1 was set as final. This means that it will override any other more specific policy. If the example above is executed again, the researcher will succeed in calling the *getData* method of patient in *ic*, as the screenshot below shows:



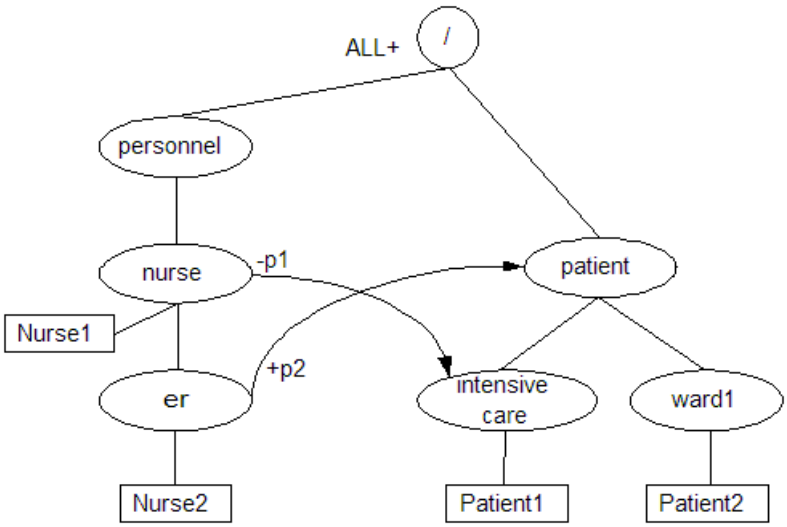
See also: Scenario1 and Scenario3

Scenario2 (last edited 2008-01-11 13:11:07 by KevinTwidle)

Scenario3

Scenario 3: crossing policies

This scenario captures the case in which an action on a specific domain of targets must not be allowed to a general domain of subjects. However, a more specific domain of subjects are allowed to execute the same action on a general domain of targets. For an example, consider the domain structure shown below:

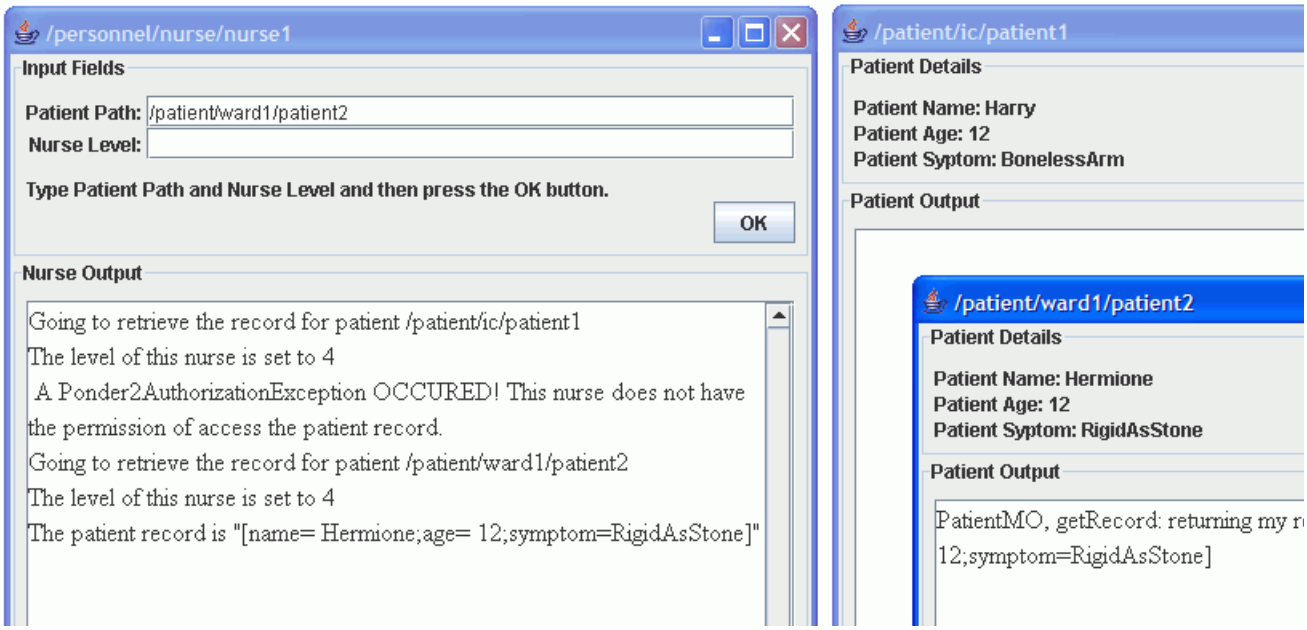


In more details, we have that nurses cannot access the patients in *ic* domain. However, nurses in emergency room (represented by the *er* domain) can have access to medical records for all the patients and in particular for patients in intensive care.

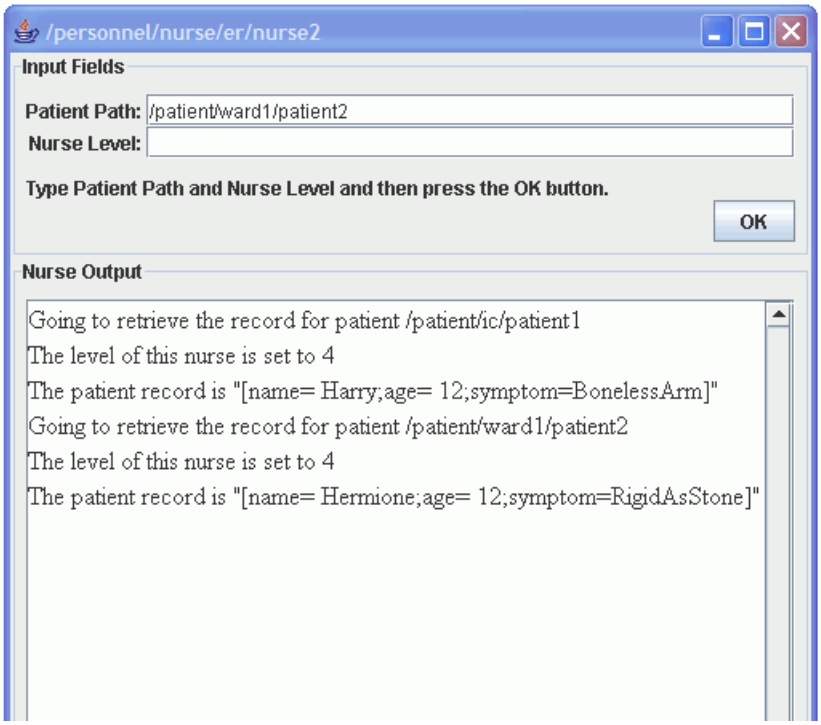
To execute this example the following command line must be used:

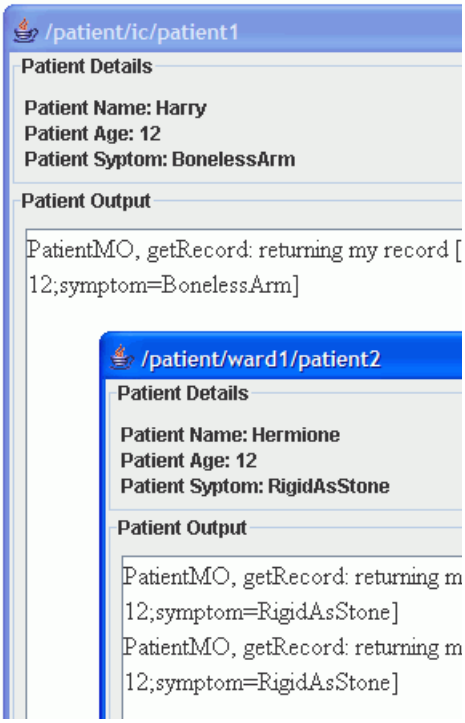
```
-auth allow  -boot hospital/hdomain.p2  -boot hospital/scenario3/nurse-settings.p2 -boot
hospital/scenario3/patient-settings.p2
```

If we send the request to Patient1 using Nurse1 then the operation will be not authorised. However, the same operation can be executed if we send the request (using the same nurse instance) to Patient2. The screenshot below shown this case:



On the other hand, using Nurse2 we can successfully send the request to both patient instances, as shown below:





See also: Scenario1 and Scenario2

Scenario3 (last edited 2008-01-11 13:11:39 by KevinTwidle)

Ponder2Shell

Ponder2 Shell

Ponder2 has a built-in shell to allow a user to interact with the SMC using simple commands.

The shell acts in some way similarly to the Unix Bourn Shell, you can move around the domain hierarchy using the change domain (*cd*) command, you can list things with the *ls* command etc. etc.

The syntax for shell commands is:

```
<shellcommand> [<arg>]+  where <command> is an internal command or PonderTalk
```

You can typically start a shell session by using telnet to the SMC. The default port number is 13570 but this may be changed by using the *-port* command option:

```
telnet localhost 13570
```

Internal Commands

Internal commands are built into the shell and can be used to manipulate the domain structure and to look at managed objects within the system.

Command	*Syntax*	*Name*	*Description*
cd	cd [pathname]	change domain	sets the current domain to the *pathname*. If no argument then goes to the root domain
ls	ls [-l] [-p] [pathname]*	list	lists contents of domains or objects. -l give more detail, -p lists policies applying to a managed object
mkdom, mkdir, md	mkdom pathname	make domain	creates and adds a new domain to the domain hierarchy
event	event name [arg]*	create event	instantiates an event type and sends it into the system. The name is interpreted as an Event Type relative to the current domain or within the '/event' domain
lp	lp [policy]+	list policy	lists the contents of a policy in XML form
ln	ln fromname toname	link	includes the *fromname* managed object in the *toname* domain

PonderTalk Statements

If the command is not recognised by the shell to be an internal command, the shell assumes that it is PonderTalk and stores up input until it gets a line ending with a full-stop (period). That input is then compiled and executed by the interpreter. PonderTalk variables are maintained for the complete session that the shell is active so a variable can be set and then used at a later time. Note, however that in PonderTalk statements all path names must start with `root` or a variable, `/domain1/domain2` for example is not accepted.

Example usage

```
$ ls
event/
factory/
shell/
$ ls event
colourevent
$ event colourevent "red" 45
...
$ var := root/mydom/myobj list.
```

```
$ root at: "mylist" put: var.
```

CategoryUsingPonder2

Ponder2Shell (last edited 2008-10-11 14:24:43 by KevinTwidle)

Ponder2 Communications

Contents

The Ponder2 SMC is capable of exporting Managed Objects from one SMC and importing them into another. Actually a reference is imported and any messages sent to the imported reference are sent as messages to the remote object. Similarly, replies from the remote object are returned to the local initiator of the messages. These message interactions are performed completely transparently with no setup being required by the user. Thus, you cannot tell whether a Managed Object is a local or remote one.

The Ponder2 SMC knows that Managed Objects may be remote but it has no knowledge of any protocols that may be used to pass the messages to and fro, external libraries need to be given on the classpath to so that protocol modules may be dynamically loaded as required. In the standard release, `ponder2comms.jar` contains some protocols that Ponder2 may use for inter-Ponder2 communications. Ponder2 knows nothing about protocols until it comes across a URL containing an unknown protocol either because it is started with the *-address* option or a remote object is specified using a URL. If the SMC does not recognise the protocol, and the first time it definitely will not, the SMC will search the classpath to see if an appropriate protocol module exists, if so it is loaded and used.

If you wish to use a protocol that has not been supplied in `ponder2comms.jar` then it is a simple matter to write your own protocol module for inter-Ponder2 communications. Once you have done so you just have to include your jar file in the Ponder2 classpath and it will be loaded when necessary, no configuration or changes to the Ponder2 core software are required.

Description

Communications protocols are divided into two sections, a transmitter for sending requests to another Ponder2 application and a receiver for receiving requests from remote Ponder2 applications. There are two operations that Ponder2 communications supports:

`getObject`	takes a pathname and returns a remote object. This is invoked by the following PonderTalk example: `root import: "root/dom/object" from: rmi://localhost/ponder2` which will load the Java RMI protocol module and use RMI as the communications channel.
`execute`	takes the unique id (OID) of a remote object, an operation and its arguments that are to be sent as a message to that remote object. The resulting object is returned. This operation is invoked by any use of a remote object that has already been imported using the `getObject` operation.

The transmitter has to pack up the arguments and transmit them to the remote end. The remote end has to receive the packed arguments, unpack them and call the equivalent recieve `getObject` and `execute` methods to perform the desired operations. The methods that perform all the work of locating the managed object and sending the messages are already included in Ponder2 and do not need to be written as part of the protocol.

Current Protocols Supplied

There are two protocols currently included in Ponder2Comms, they are, Java Remote Method Interaction (RMI) and HTTP Web Services.

RMI

Before using RMI communications in Ponder2, the Java RMI registry must be running on all the computers that Ponder2 is to run on.

When invoked, the RMI receive class registers itself with the Java RMI registry to enable it to receive RMI calls. These calls will occur asynchronously to the current operations of Ponder2.

To use RMI, the `java.rmi.server.codebase` property value must be set correctly. It represents one or more URL locations from which the RMI stub code used by the Java RMI system (and any classes needed by the stubs) can be downloaded. This property must be set to a URL which can be an `http://` URL or a `file:///full/pathname/...` URL. If multiple JAR files are to be specified, as is the case here, then the URLs are separated by a space. As usual, if there is more than one argument then quotes are required to keep them together. The Ant `build.xml` command file

already sets up the codebase correctly so you need not worry about it if you are using this file.

To make sure that Ponder2 registers itself with a particular name, the *-address* option can be given when it is started[1]

```
  java -Djava.rmi.server.codebase="file:///full/pathname/ponder2.jar file:///full/pathname
/ponder2comms.jar"\
       -classpath ponder2.jar:ponder2comms.jar net.ponder2.SelfManagedCell -address
rmi://localhost/MyPonder
```

The *address* argument may be repeated to register Ponder2 under several different names at the same time. See -address for more information.

Errors

Note: The following error indicates that `ponder2comms.jar` cannot be located by RMI via the codebase property.

```
RmiProtocol cannot create Ponder2: RemoteException occurred in server thread; nested
exception is:
        java.rmi.UnmarshalException: error unmarshalling arguments; nested exception is:
        java.lang.ClassNotFoundException: net.ponder2.comms.rmi.RMIReceiveInterface
```

Note: It is possible that you will get a *class not found* error involving `Element` or `TaggedElement` in which case you will have to add `qdparser.jar` to the codebase argument.

SSComms

Single Socket Communications. This is a protocol using a simple server that allows a Ponder2 SMC to create a single TCP socket and then send and receive messages asynchronously over it. This allows a device, like a phone, that can only make outgoing TCP connections to participate fully in a distributed environment. Messages send to the SMC will be received over the same connection that outgoing messages are using.

SSComms requires a server that must be run on an accessible host computer. Every SMC taking part in the distributed system makes a connection to the server and all communications takes place to and from the server with the server routing the messages based on the addresses being used. the server is started as:

```
  java -jar sscomms.jar
```

This will start the server and will log all messages being sent through it. You can type a 't' command to it and that will turn tracing on or off so you can see the contents of the messages passing through.

SMC clients use the `ssc` protocol. e.g.

```
  java -classpath ponder2.jar:ponder2comms.jar:sscomms.jar net.ponder2.SelfManagedCell
-address ssc://hostip/myphone
```

This will start an SMC which will open a socket to a server on the given host `hostip` and register itself as `myphone`. If another SMC want to communicate, e.g. import an object, then it can do:

```
  remoteobj := root import: "ssc://hostip/myphone".
```

The *address* argument may be repeated to register Ponder2 under several different names at the same time. See -address for more information.

HTTP/Web Services

HTTP Web Services only includes a transmitter class for making Web Service calls. Calls are received via Web Services by a web server which then forwards them to Ponder2 using RMI; thus no state is required at the web server for Ponder2 communications.

A special Java Web Service (.jws) file is included in the Ponder2 Comms distribution and also attached here at Ponder2.jws. This file can be used by Apache Axis under Apache Tomcat to provide the appropriate Web Service interface to Ponder2 communications.

To run Ponder2 using HTTP and Axis you must configure Axis

1. Copy the Ponder2.jws file to AXIS_HOME
2. Copy ponder2.jar and ponder2comms.jar to AXIS_HOME/WEB-INF/lib
3. Restart Axis

There is a detailed step by step description in Ponder2 with Axis.

You must also include all the following files in your classpath:

```
ponder2.jar
ponder2comms.jar
commslib/activation.jar
commslib/axis.jar
commslib/commons-discovery-0.2.jar
commslib/commons-logging-1.0.4.jar
commslib/jaxrpc.jar:commslib/mail.jar
commslib/saaj.jar
commslib/wsdl4j-1.5.1.jar
```

The Ponder2.jws file communicates with Ponder2 using Java RMI and expects the registered name to be *Ponder2* therefore Ponder2 must be started with the option

-address rmi://localhost/Ponder2

If you want to use a different name then change the .jws file and restart Axis.

Managed objects can be imported from this Ponder2 instance using:

Toggle line numbers

```
1 root import: "root/an/object" from: "http://www.host.com:8080/axis/Ponder2.jws"
```

Rolling your own

To add your own protocol it is probably a good idea to look at the current code for RMI communications. The class named net.ponder2.comms.RmiProtocol has a special name, the others can be called anything and are grouped into the package net.ponder2.comms.rmi for convenience.

Class	*Description*
net.ponder2.comms.RmiProtocol	When the RMI protocol is first mentioned in a URL e.g. rmi://localhost/Ponder2 this is the class that is located dynamically by Ponder2. It *must* have this name. The name is calculated from the scheme, in this case "rmi" Note the capital letters, the first letter of the scheme is capitalised as is the first letter of the word "Protocol". This class knows of the classes mentioned below and makes use of them. None of the other classes have special names
net.ponder2.comms.rmi.RMIReceiveInterface	Needed by the RMI protocol for communications
net.ponder2.comms.rmi.RMIReceiver	The receiver, registered with the RMI registry by the class RmiProtocol, that waits for RMI communications
net.ponder2.comms.rmi.RMITransmitter	The transmitter registered with Ponder2 by the RmiProtocol class. This is used to send inter-Ponder2 requests

Let's say we want to write a protocol for Appletalk. We will use URLs of the form atalk://an/appletalk/specific /address. When a URL of this form is used, Ponder2 will look for a class called net.ponder2.comms.AtalkProtocol. This class simply contains an install method which may perform some protocol specific initialisation and it may set up a receiver in another thread. It must then register the transmit side of the protocol into Ponder2. e.g.

Toggle line numbers

```
1 public class AtalkProtocol implements net.ponder2.comms.Protocol {
2   public void install(URI address) {
3     // Perform protocol specific initialisation
4     initialiseAtalk();
5     // Start a receive thread for the protocol
6     new ATALKReceiver(address);
7     // Now register the protocol and let the external managed object use it
8     ExternalManagedObject.registerProtocol("atalk", new ATALKTransmitter(), address);
9   }
10 }
```

The argument to *install* is the address that this Ponder2 application will be know by to the outside world using this protocol i.e. our address using the protocol to be installed. The arguments to registerProtocol (API http://InDocHereSomewhere) are:

`String name`	The name of the protocol
`ExternalCommsProtocol transmitter`	An instance of the protocol transmitter so that messages can be sent to another Ponder2 application.
`URI address`	The address so that it can be embedded in OIDs so that given an OID any Ponder2 application can communicate with our Ponder2 application using the protocol that is being loaded.

As mentioned above Ponder2 uses *getObject* and *execute* to communicate. In addition, the transmitter instance has a *connect* method which is called every time a new address using that protocol is to be used. *Connect* can act as a factory method returning a new instance of the transmitter if state is required for each remote address, or it can return itself if address resolution is to be performed every time *getObject* and *execute* are called.

Toggle line numbers

```
1 public interface Transmitter {
2
3   /**
4    * creates and connects a Transmitter to a remote location. This is called
5    * once every time a new remote address is brought into play. If the location
6    * is important to the Transmitter i.e. a permanent channel is opened then a
7    * new instance of transmitter should be created. If the remote location does
8    * not matter then only one instance of the Transmitter need be created and
9    * this method can return itself.
10   *
11   * @param address
12   *            the location that this protocol is to be connected to
13   * @return a new communications protocol connected to the appropriate place or
14   *         null if it fails
15   */
16  public Transmitter connect(URI address);
17
18  /**
19   * gets a managed object from a remote SMC
20   *
21   * @param location
22   *            the address of the remote SMC
23   * @param path
24   *            the full path name of the remote managed object
25   * @return the result of the operation.
26   * @throws Ponder2Exception if an exception occurs in the remote system
27   */
28  public P2Object getObject(URI address, String path) throws Ponder2Exception;
29
30  /**
31   * executes commands at a remote managed object. The command will either be a
32   * create or use clause.
33   *
34   * @param address
35   *            the address of the remote SMC
36   * @param target
```

```
 *            the remote object's OID
 * @param source
 *            the originator of the operation
 * @param op
 *            the operation to be performed
 * @param args
 *            the arguments for the operation
 * @return the result of the operation
 * @throws Ponder2Exception if an exception occurs in the remote system
 */
 public P2Object execute(URI address, OID target, P2Object source, String op, P2Object[] args)
      throws Ponder2Exception;
}
```

The information in the arguments **must** be transferred, using the protocol, to the remote Ponder2 application (marshalling). At the receiving end, the arguments must be reconstructed and the supplied receive methods can be called to do all the work(unmarshalling).

A suitable transmitter for AppleTalk could be:

Toggle line numbers

```
package net.ponder2.comms.atalk;

import java.net.URI;

import net.ponder2.P2Object;
import net.ponder2.comms.Transmitter;
import net.ponder2.exception.Ponder2Exception;

/**
 * This class embodies the sending mechanism for inter-SMC communications using
 * the AppleTalk protocol.
 *
 * @author Kevin Twidle
 */
public class ATALKTransmitter extends TransmitterImpl implements Transmitter {

  public Transmitter connect(URI location) {
    // We will use the location whenever we send something
    // otherwise we would return new ATALKTransmitter(location)
    return this;
  }

  public P2Object getObject(URI location, String path) throws Ponder2Exception {
    // We assume that the AppleTalk protocol can only send and receive a single String
    // So pack all the arguments into a single XML structure and sent it as a String
    // We will receive an XML string containing a reply object
    // We will use the supplied helper routine which will call execute to perform the transfer
    return getObjectString(location, path);
  }

  public P2Object execute(URI address, OID target, P2Object source, String op, P2Object[] args)
    throws Ponder2Exception {
      return executeString(address, target, source, op, args);
  }

  protected String execute(URI address, String xmlString) throws Ponder2Exception {
    try {
      return sendThisAtalkMsg(location, xmlString);
    }
    catch (AtalkRemoteException e) {
      throw new Ponder2OperationException(e.getMessage());
    }
```

```
  }
}
```

On the receiving end we need something like:

Toggle line numbers

```
// Called when AppleTalk receives a message
// This assumes a request - reply mechanism

public TaggedElement getObject(URI location, String path) throws Ponder2Exception {
  // Receiver.execute() may call this routine
  return Receiver.getObject(location, path).writeXml();
}

public String execute(String sxml) throws Ponder2Exception {
  // Make use of XML string helper code, it will decode and execute the call.  It will
  // encode the result as a String an return the result.
  return Receiver.execute(sxml);
}
```

CategoryPonder2Programming CategoryPonder2Installation

1. use ; in the classpath for Windows (1)

Ponder2Comms (last edited 2010-03-23 16:55:53 by KevinTwidle)

ExternalCommunications

External Communications

One of the reasons for using Ponder2 is to provide a policy framework for other, unrelated applications. This means that Ponder2 has to receive information from other applications in order to make its decisions and to communicate those decisions back to the application.

Contents

Sending Information

Receiving Information

Information can come into the Ponder2 SMC as either Events or Commands

Receiving Events

Receiving Commands

There is a useful Managed Object called PonderTalk which accepts PonderTalk statements either as arguments or over an RMI connection that it maintains. Documentation can be found in the Ponder2 release tree at `doc/pondertalk/PonderTalk.html`.

Examples

Simple RMI Example

The PonderTalk managed object can easily be used to send and receive PonderTalk commands from external sources via RMI. First the PonderTalk module must be set up to listen to an RMI port, in this case we are using one called "MyPonder2".

Toggle line numbers

```
factory := root load: "PonderTalk".
pt := factory create: "MyPonder2".
```

The RMI connection can be tested internally with the following. You should see "This is a test" written to the console.

Toggle line numbers

```
pt test: "root print: \"This is a test\"".
```

Now we can try it from afar, as far as a shell anyway. To a shell prompt enter

```
$ java -cp lib/ponder2.jar net.ponder2.PonderTalk 'root print: "Hello, World!".'
```

you should see *Hello, World!* printed out on the Ponder2 console.

The RMI execute interface used takes a simple string, this is an extract from the PonderTalkInterface class. The proper one from the ponder2.jar file should be used - net.ponder2.PonderTalkInterface.

Toggle line numbers

```
public interface PonderTalkInterface extends Remote {

  /**
   * Takes a PonderTalk string, compiles and executes it. Returns the result as
```

```
 * a string.
 *
 * @param ponderTalk
 *            a string containing one or more PonderTalk statements separated
 *            by full-stops (periods).
 * @return the result of the operation as a string
 * @throws RemoteException
 *             if something goes wrong
 */
public String execute(String ponderTalk) throws RemoteException;
...
...
}
```

The code used in the PonderTalk main to send the RMI message is copied below and can be incorporated into your external Java code.

Toggle line numbers

```
public static void main(String args[]) {
    String result;
    boolean ok = true;
    if (args.length < 2)
        System.out.println("Too few arguments. Need: RMIname pondertalk statement.");
    String rmiName = args[0];
    String ponderTalk = "";
    for (int i = 1; i < args.length; i++) {
        ponderTalk += " " + args[i];
    }
    try {
        PonderTalkInterface pt = (PonderTalkInterface) Naming.lookup(rmiName);
        result = pt.execute(ponderTalk);
    }
    catch (Exception e) {
        result = "Remote PonderTalk failure: " + e.getMessage();
        ok = false;
    }
    System.out.println(result);
    System.exit(ok ? 0 : 1);
}
```

For a comprehensive description of using external communications see thexmlBlasterexample.

ExternalCommunications (last edited 2008-09-23 13:03:36 by KevinTwidle)

Ponder2 Internals

This page introduces the inner-workings of the Ponder2 SMC and guides the programmer through some of the code.

- Ponder2 Class Structure The basic class structure of Ponder2
- Ponder2 Bootstrap How the system starts up
- Ponder2 Internal XML The layout of the XML produced by the PonderTalk compiler
- Ponder2 Event Bus Ponder2's Proximity Event Bus

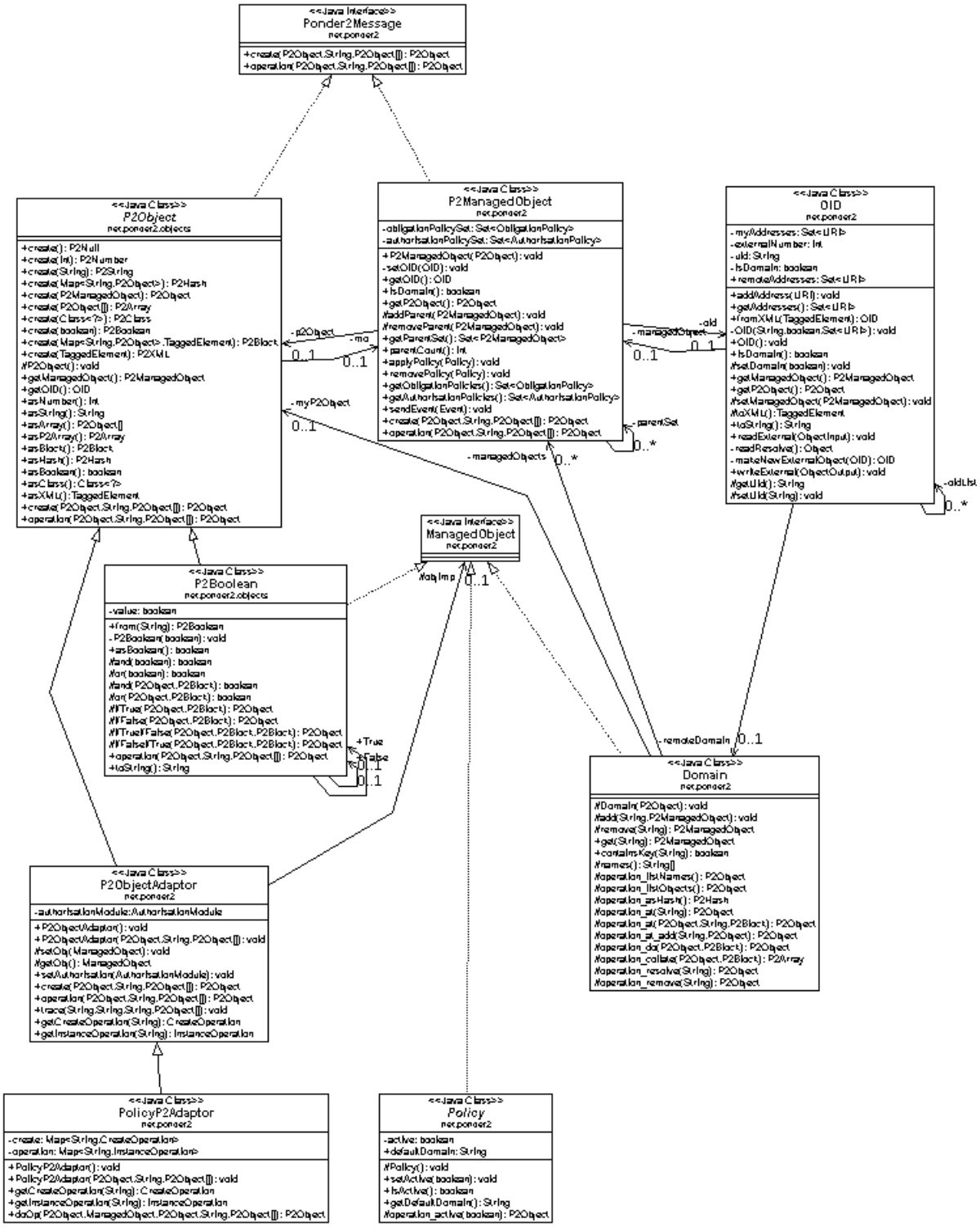
Ponder2Internals (last edited 2008-08-01 15:26:05 by KevinTwidle)

Ponder2Classes

Ponder2 Class Structure

This section of the Ponder2 Wiki describes the internal core Java classes that comprise the SMC. This information is designed to be read in conjunction with the Ponder2 JavaDoc documentation and the Ponder2 Managed Object documentation.

The basic set of classes is shown in the following UML diagram



Overview

All Java managed object code files (including user code) implement the empty interface `ManagedObject`. This tells the compiler that it is to be a managed object. e.g. Policy and Domain do this.

All Java Managed Objects get a shadow sub-class of P2ObjectAdaptor which provides the calls into the user code. e.g. PolicyP2Adaptor is shown. It contains the PonderTalk commands that can be made to a Policy. This is generated by the compiler.

P2ObjectAdaptor and basic objects (e.g. String, Boolean etc.) are all subclasses of P2Object.

P2Object is the main API for objects in the system.

Domains now hold instances of P2ManagedObject. A P2ManagedObject is only created for a P2Object when required. That is to say, the overhead of creating the tables and OID for an object is only performed when e.g. the object is to be put into a domain. If the object is used only in variables or as arguments at the PonderTalk level then the overhead is avoided. This speeds up creation of objects.

OIDs are only used by the system when managed objects are external to the Ponder2 system. Every P2ManagedObject has an OID. OIDs refer back to the P2Managed object unless the OID is for an external object.

Ponder2Classes (last edited 2008-01-03 17:40:14 by localhost)

Ponder2Bootstrap

Ponder2 Bootstrap

The Ponder2 SMC comprises a completely empty root domain when it starts up. The bootstrap code below is executed to give the initial structure to avoid the user having to write it for each application. The PonderTalk code is followed by the bootstrap that was used by the previous version of Ponder2 when XML was used as the configuration/command language. Both sets of code are fully commented; it can be seen that the new code is considerably more readable than the old.

The option `-boot` – can be given to Ponder2 when it is started to prevent the loading of this bootstrap code.

Toggle line numbers

```
// Bootstrap code for Ponder2

// Import the Domain code and create the default domains
root add: "domaintemplate" obj: ( root import: "Domain" );
     add: "template" obj: root/domaintemplate create;
     add: "policy" obj: root/domaintemplate create;
     add: "event" obj: root/domaintemplate create.

// Move the domain template into the template directory
root/template add: "domain" obj: root/domaintemplate.
root remove: "domaintemplate".

// Import event and policy templates
root/template add: "event" obj: ( root import: "EventTemplate" );
              add: "policy" obj: ( root import: "Policy" ).
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- First import a Domain type, create a Template domain and move the Domain type to the new
domain -->
<xml>
        <!-- Import the template for creating domains -->
        <use name="/">
                <!-- /.add("domaintemplate",import(Domain) -->
                <add name="domaintemplate">
                        <use name="/">
                                <import name="Domain" />
                        </use>
                </add>
                <!-- /.add("template",/domaintemplate.create()) -->
                <add name="template">
                        <use name="/domaintemplate">
                                <create />
                        </use>
                </add>
        </use>
        <!-- Link the Domain template into the new domain -->
        <use name="/template">
                <!-- /template.link("domain", "/domaintemplate") -->
                <link name="domain" object="/domaintemplate" />
        </use>
        <!-- Unlink the original temporary template -->
        <use name="/">
                <!-- /.remove(domaintemplate) -->
                <remove name="domaintemplate" />
        </use>
        <!-- Create the framework domains -->
        <use name="/">
                <!-- /.add("policy", /template/domain.create()) -->
```

```
                <add name="policy">
                        <use name="/template/domain">
                                <create />
                        </use>
                </add>
                <!-- /.add("event", /template/domain.create()) -->
                <add name="event">
                        <use name="/template/domain">
                                <create />
                        </use>
                </add>
        </use>
        <!--  We need to create events -->
        <use name="/template">
                <!-- /template.add("event", import(policy.EventTemplate)) -->
                <add name="event">
                        <use name="/">
                                <import name="EventTemplate" />
                        </use>
                </add>
        </use>
        <!-- We need to create policies -->
        <use name="/template">
                <!-- /template.add("event", import(policy.Policy)) -->
                <add name="policy">
                        <use name="/">
                                <import name="Policy" />
                        </use>
                </add>
        </use>
</xml>
```

Ponder2Bootstrap (last edited 2008-02-12 10:00:17 by KevinTwidle)

Ponder2 Internal XML

PonderTalk is compiled to an intermediate XML. At run-time the XML is interpreted. The XML is taken as a series of XML elements, each one returning a Ponder2 managed object (P2Object). The XML elements comprise one of:

Contents

XML	*Tag Description*
`ponder2`	Dummy element used to enclose many elements into one to meet the XML document standard. Returns the result of the last element
`assign`	Used to assign an object to a variable. Returns the object assigned
`send`	Send one or more messages to a managed object. Returns the result of the last message sent
`use`	Uses a variable or resolves a path name. Returns the variable value or the object located
`string`	Returns an object containing a string value
`boolean`	Returns an object containing a Boolean value
`number`	Returns an object containing a numeric value. Java BigDecimal is used here to give accurate, arbitrary precision values
`block`	Returns an object containing a block of XML code for later execution
`array`	Returns an object representing an array of objects
`oid`	Returns an object located by its unique identifier and address. If the object is remote then a local External Managed Object is returned as a proxy

Table of possible XML elements

Send

The XML send structure essentially identifies a managed object and sends one or more messages to it. The result is the return values of the last message.

```
<send>
 <object/>
 <message name="message_name" …>
   <argument/><argument/>…
   </message>
 <message …> …
 </message> …
</send>
```

Where <object/> is one of the XML elements in the above table. Since everything returns an object in PonderTalk, the result of <send> is an object, thus allowing a <send> element to be used in place of <object/>. This gives a very simple yet very powerful XML structure that is general enough for all the decoding work to be done before handing the message to the Java managed object.

Block

OID

Array

Use

Assign

Ponder2InternalXML (last edited 2008-02-06 13:22:06 by KevinTwidle)

Ponder2 Tutorial

Contents


```
Please note: This version of the Ponder2 Tutorial is dated 23rd
May 2009.
```

The Ponder2 Tutorial zip file contains a complete Ponder2 system with extensive tutorial pages, examples and exercises. It can be used on Windows, Linux and Mac OS X. The tutorial has an Apache ant configuration that will compile and run the exercises. A recent copy of Apache ant (1.7) is included and will be used automatically by the tutorial if needed.

After downloading the zip file, extract the files and open `index.html` in the top level directory.

Please note that the Jar files included in the tutorial are not necessarily the most recent ones available. After going through the tutorial, you are encouraged to download the latest files from the download page.

Note for Windows users: If you are using Windows it is better to open the zip file and then drag the top-level directory (Ponder2Tutorial) to your desktop. If you use the "Extract to here" function then dates and times seem to get lost and *ant* will rebuild the system the first time you run it, even though it is not necessary.

Changes since the last version

Fixed the incorrect RMI names for Exercise 1. The basic BSN demo did not work because the BSN sensors had the wrong RMI names.

Ponder2Tutorial (last edited 2009-05-23 17:37:31 by KevinTwidle)

Ponder2Glossary

Glossary

This pages contains definitions of terms used elsewhere in this site. It is maintained in alphabetical order, please keep it that way when adding definitions.

Contents

Command Interpreter

The command interpreter interprets the XML received by Ponder2. See Ponder2 XML for the detail of these commands.

Domain

Domains are a means of grouping Managed Objects in an hierarchical structure for management purposes. The grouping is explicit i.e. objects have to be explicitly included/removed from domains. Domains are managed objects themselves. Note that domains only contain links to the managed objects not the managed objects themselves. Thus, a managed object can be member of multiple domains.

Domain Service

The Domain Service maintains the parent/child relationships of domains and other managed objects.

Managed Object

A Managed Object is an entity in Ponder2 capable of receiving and replying to PonderTalk messages. A Managed Object is written in Java and uses Java annotations (i.e. `@Ponder2op()`) to create the links between PonderTalk message keywords and Java methods. A Managed Object may communicate with other managed objects, it may use Swing as part of a GUI, it may act as an adaptor and communicate with external entities etc. etc.

This simple Managed Object guide gives a step by step explanation for designing, implementing and running a new Managed Object.

This more advanced guide explains how to increase the functionality of your Java Managed Objects such as sending messages to other Managed Objects.

Obligation Policy Interpreter

When the occurrence of an event is notified to Ponder2, the obligation policy interpreter matches the notification against the registered policy events, hands the event to the relevant policies which then evaluate their condition(s) and if they succeed, invoke the action(s) specified in the policy.

Self-Managed Cell

A self-managed cell (aka SMC) is defined as a set of hardware and software components forming an administrative domain that is able to function autonomously and thus capable of self-management.

Ponder2Glossary (last edited 2009-11-18 17:22:39 by KevinTwidle)

ToDo

Notes for future changes, enhancements and bug fixes

- Allow conditions and actions to be cleared on ECA policies
- Fix obsolete ECA documentation

ToDo (last edited 2012-01-08 20:16:00 by KevinTwidle)

Teleo Reactive Ponder2

Contents

Introduction

A teleo-reactive (T-R) program is a mid-level agent control program that robustly directs an agent toward a goal in a manner that continuously takes into account the agent's changing perceptions of a dynamic environment. T-R programs are written in a production-rule-like language and require a specialized interpreter [ref: nilsson].

T-R programs typically have an ordered set of rules (a Rule Set) each of which contains a condition and one or more actions. The Rule Set then selects the first rule whose condition is true and starts executing the rule's actions. What follows afterwards depends upon the Rule Set's configuration and is described below.

Example

Rule	*Condition*	*Action*
1	unloaded & at depot	finished
2	loaded & at depot	unload
3	loaded & facing depot	go forwards
4	loaded	turn left
5	next to bin	load bin
6	facing bin	go forwards
7		turn left

Here is a simple example of a T-R rule set. It is for a robot truck that starts somewhere other than its depot. It's goal is to pick up a bin and put it down at the depot. When it starts off it may or may not be facing the bin (we assume that it is not next to it but that would work too). We can see that the first five rules' conditions are not satisfied, so we look at rule 6. If the truck is not facing the bin then rule 6 would be invoked. If the truck is facing the bin then rule 7 would be invoked and it would move forwards towards the bin.

If the rule set were continuously evaluated you can see that the truck would start turning to the left, then move forwards, pick up the bin, turn to the left again, go forwards and drop the bin off at the depot. This is because the lower numbered rules take precedence over the higher numbered ones. So, if rule 7 were being continuously executed then as soon as the truck were facing the bin, rule 6 would take over and the truck would move forwards.

Note that if someone moved the bin while the truck was approaching it (using rule 6) then rule 6's condition would no longer be true and rule 7 would be executed continuously until the truck were facing the bin again.

Example Notes

- Rule 7 is executed if the truck is not loaded and it is not at the depot and it is not facing the bin. It does not require a condition (empty implies *true*) because if none of the other rules' conditions are satisfied we must run this rule.
- It would be more efficient if we had two rules with conditions asking if the bin were to the left or the right of the truck and actions to turn in the appropriate direction.
- Another enhancement would be to have rule 6 running in parallel with a turning rule so the truck drive forwards and turn at thesame time towards the bin.

Rules

Rules have a condition and one or more actions. The rule can be asked if it is available to be run i.e. to evaluate its condition and return the result, true or false. The rule can also be told to execute its actions. A rule can handle two or more parallel sequences of actions, it does this by spawning off separate threads and waiting for them all to finish before indicating that its actions have been completed. Actions are normally blocks but may be Rule Sets or even other rules, more about this later, for the time being we will just use blocks. **NB** Conditions are likely to be run many times and at any time therefore condition blocks should not have any side effects.

A rule in PonderTalk is created as follows:

Operation	*Description*
condition: aCondition	Creates a rule with the given condition block aCondition
condition: aCondition action: anAction	Creates a rule with the given condition block aCondition and the given action action anAction
condition: aCondition actions: anActionArray	Creates a rule with the given condition block aCondition and the given array of actions in anActionArray

Example: Creating a new rule

Here we can see the T-R rule factory being imported and then a new rule Managed Object is created with a condition that tests if myValue equals 23. Its action is a block that when activated moves the cart forwards. Notice that both the condition and action are blocks. Blocks are not executed when they are compiled but are activated later when needed; in this case, when the rule is told to evaluate its condition and when the rule is told to run it action(s).

Toggle line numbers

```
1 rule := root load: "TrRule".
2 myrule := rule condition: [ myValue = 23 ] action: [ cart forwards ].
```

Once the rule has been created, more actions can be added to it, including parallel markers. The operations allowed on a rule are:

Operation	*Description*
action: anAction	Adds an action to the rule
actions: anActionArray	Adds an array of actions to the rule. The actions are copied from the array, the array is discarded
parallel	Indicates that any following actions are to be run in parallel with the previously given actions. This command may be used many times for multiple parallelism

Example: Adding to a rule

In this example actions are added to the rule. Blocks can be defined and held in domains or variables so these can be to reference them. We show here some blocks being created and then being given to the rule. **NB** Rules are not (yet) themselves thread safe so any one instance of a rule can only be used at any one time. Do not give the same rule to different rule sets.

Here we can see that two threads will be started when the rule is run. One thread will move the cart forwards, left, right and then stop it. The other thread will, independently, make the cart go fast, then slow, then fast again.

Toggle line numbers

```
1 checkValue := [ myValue = 23 ].
2 root/actions at: "forwards" put: [ cart forwards ]
3 myrule := rule condition: checkValue.
4 myrule actions: #( root/actions/forwards [cart left] [cart right] ).
5 // The following line adds an action to the current action sequence, after [cart
right]
6 // then it starts a new, parallel action sequence that will be run in parallel to the
first 4 actions
7 myrule action: [ cart stop ]; parallel; action: [ cart fast. cart slow. cart fast ].
```

The following examples are the same. The rule runs blocks *b1*, *b3* and *b4* in parallel. *b2* is run when *b1* finishes. *b5* is

run when *b4* finishes. The rule finishes when all parallel threads have finished.

Toggle line numbers

```
1 myrule action: b1; action: b2; parallel; action: b3; parallel; action: b4; action: b5.
2 // The following is the same as the preceeeding line
3 myrule actions: #( b1 b2 ); parallel; action: b3; parallel; actions: #( b4 b5 ).
4 // The following is similar but not identical
5 myrule action: [ b1 value. b2 value ]; parallel; action: b3; parallel; action: [ b4
value. b5 value ].
```

Stopping Rules

Rules can be stopped using the *stop* command. This is normally done by the system not the user. However the user should be aware how the rule manages stop commands. If a stop is received, sends a stop to the currently running action(s) and then waits for them to terminate. Once they have finished the rule returns control to the requester. Blocks are atomic as far as PonderTalk is concerned and cannot be interrupted (and therefore don't receive the stop). This should be taken into account when designing the rule's actions.

In the above example, if either of the first two rules are used then a stop could cause the rule to terminate without having run *b2* or *b5* or even both. In the third example, if the rule receives a stop command then the rule will wait until all three threads have run to completion because blocks are atomic. I.e. even if *b2* hasn't started it will still be run when *b1* finishes before the action finishes.

Rule Sets

Rule Sets are the heart of T-R programming. A rule set is given an ordered set of rules which it then manages. The rule set, when started, asks each rule in turn if it can be run, the first rule to respond that it can, gets run. The rule set then proceeds from the top evaluating the rules over and over again. Each time the rules are evaluated, the top-most rule is run. If the top-most rule is the currently running rule (and it is still running) then that rule is simply left alone to continue. If, however, the highest priority rule ready to run is not the currently running rule then the currently running rule is told to stop; when it has stopped, the chosen rule is run.

Creating a Rule Set

A rule set in PonderTalk is created using one of the following messages:

Operation	*Description*
create	Creates a new TrRuleSet
create: aName	Creates a new TrRuleSet with the name aName (used for trace messages)

Here we can see the TR rule set factory being imported and then a new rule set Managed Object being created:

Toggle line numbers

```
1 ruleset := root load: "TrRuleSet".
2 myruleset := ruleset create.
```

Adding Rules to a Rule Set

Once the rule set has been created it can have rules added to its ordered sequence of rules. Rules can also explicitly be added at a certain position. Once everything is set up the rule set can be told to start whilst giving it the percepts hash:

Operation	*Description*
add: aRule	Adds aRule to the end of the ordered rule set.
at: anIndex **put:** aRule	Adds aRule at position anIndex to the rule set. Displaced rules get pushed down by one

In this example some rules are created and added immediately to a rule set:

Toggle line numbers

```
1 ruleset add: (rule condition: [ c > 4 ] action: [ a := a + 1 ]).
2 ruleset add: (rule condition: [ a > 4 ] action: myAction).
```

```
// Whoops we forgot a rule(!), let's add it in front of the last one
ruleset at: 1 add: (rule condition: [ true ] action: anotherAction).
```

Starting and Stopping Rule Sets

A rule set is started by sending it a run command along with the percepts that the rule set and all its rules will be using. The start is asynchronous, a new thread is started for the rule set. The stop is synchronous i.e. the stop does not return until the rule set has stopped its currently running rule. A rule set can explicitly be told to re-evaluate its rules by sending it an explicit event (this is how the percepts tells the rule set to re-evaluate the rules' conditions):

Operation	*Description*
run: thePercepts	runs the ruleset using the percepts found in thePercepts. The rule set is run in a separate thread so that it can be given new events or may be stopped. This call immediately returns leaving the ruleset running.
stop	Tells the rule set to stop executing rules. If a rule is running then that rule is told to stop. This action returns after the currently running rule (if any) has been stopped

Example: Controlling a rule set

Here we can see the TR rule set being started, later being told to reevaluate its rules and then still later stopped:

Toggle line numbers

```
myruleset run: mypercepts.
// Wait for 10 seconds before telling the rule set to reevaluate its rules.
root sleep: 10.
myruleset event.
// Wait for 20 seconds before stopping the rule set
root sleep: 20.
myruleset stop.
```

Example: Controlling multiple rule sets

Multiple rule sets can be given events

Toggle line numbers

```
mypercepts tell: myruleset1; tell: myruleset2.
myruleset1 run: mypercepts.
myruleset2 run: mypercepts.
// Wait for 10 seconds before telling all the rule set to re-evaluate their rules.
root sleep: 10.
mypercepts event.
// This line will also cause the rule sets to re-evaluate their rules
percepts at: "myvar" put: "myvalue".
```

Debugging

It can sometimes be difficult to see what is happening in a T-R program, especially if there are multiple threads running. The TrPonder system has a trace facility that can be turned on and off to help keep track of the rules being run. Debugging is turned on with the TrRuleSet trace command:

Operation	**Description**
trace: aBoolean	Sets all T-R tracing on for all rule sets and rules if aBoolean is *true*. Tracing is set to *false* initially

Debugging is turned on and off for the whole TrPonder system running within an SMC. That means that all rule sets and rules will start generating tracing information to the Java console until tracing is turned off again.

Toggle line numbers

```
// Turn tracing on
myruleset trace: true.
// Turn tracing off
myruleset trace: false.
```

Rule Set Behaviour Modifications

In the strict sense of T-R programs, the Rule Set constantly re-evaluates its rules' conditions. If a rule with a higher priority (earlier on in the ordered set) than the currently running rule becomes available to run then the currently running rule is stopped and the new rule is run. This is not very efficient in a multasking system so various options have been created to manage the rule sets efficently:

Run and Wait

The Rule Set can be told to run one rule and then wait for a new event to occur or the rule to finish before re-evaluating all the rules' conditions.

Wait For Rule

The Rule Set will not try to stop a currently running rule even if there is a higher priority ready available to run.The Rule Set will wait for the rule to terminate before running the new rule.

Run Once

The Rule Set can to told to run one rule and then terminate. This implies *Wait For Rule*.

Event

The event command tells the rule set that an event has occurred and that the rules should be reevaluated. This may be overridden by one of the above settings. This call is used by the precepts to indicate that a value has changed.

These operations are performed on rule sets using the following PonderTalk commands:

Operation	*Description*
event	Tells the receiver to re-evaluate the rule conditions and to choose a (possibly) new rule to run
runOnce: aBoolean	Tells the rule set to run one rule and then terminate if aBoolean is *true*. The initial value is *false*
waitForEvent: aBoolean	Tells the rule set to wait for an event before running through the rule list again if aBoolean is *true*. Use of this option can remove unnecessary scans of the rule conditions. The initial value is *false*.
waitForRule: aBoolean	Tells the rule set to wait for completion of a rule rather than interrupting it as necessary if aBoolean is *true*. The initial value is *false*

Percepts

Percepts contain the global knowledge that the T-R rule set shares with all the rules. The rule set is informed whenever the percepts change and, depending upon the options used, this could trigger a new evaluation of the rules and therefore possibly a new rule to be run. The percepts managed object is derived from a PonderTalk Hash managed object which is a basic managed object within Ponder2. As such it needs to be created in the following way which ensures that the new object looks like a hash to other managed objects (internally, basic managed objects are treated differently from normal managed objects, see the source code for more details):

Toggle line numbers

```
pfactory := root load: "Percepts".
percepts := pfactory create asHash.
```

The percepts object can now be used as a hash in the normal way, it can also be told to notify one or more rule sets should any values be written to it. The percepts object is given to a rue set when the rule set is run. The rule set hands it to each rule that is run, the rule hands it to each action. Blocks see the percepts values a global variables so block arguments are not required.

Operation	*Description*

tell: aRuleSet	Tells the percepts that it must inform the given rule set whenever a value changes or an event is raised. This command may be used many times to inform many rule sets
event	Tells the percepts to inform the rule set(s), given by the *tell* command, that an event has occurred
at: aName **put:** aValue	Store a value and raise an event for the rule set(s)

Percepts Examples

PonderTalk blocks can take arguments from a Ponder2 hash managed object. They can also merge the values from a hash into their environment variables before executing the block's code. This allows values in a hash to be used as plain variables without declaring them as arguments.

Toggle line numbers

```
1 percepts at: "size" put: 5.
2 rule condition: [ size == 5 ] action: ...
3 // The rule runs this as:
4 ruleCondition valueVars: percepts
5 // Otherwise we would need
6 rule condition: [ :size | size == 5 ] action: ...
```

Actions

The examples above have all shown actions as blocks. In addition to blocks, actions may also be rule sets or other rules. All three types are described more fully here.

Code Blocks

A code block is a non interruptible sequence of operations. If a long running action is to be made interruptible then it should be split into a sequence of actions then after one code block as finished the rule can be terminated before the next action is started. Conversely if there are several actions that should be atomic then they should be grouped into a single code block.

Rule Sets

Other Rules

If another rule is given as an action then it is simply told to execute its actions when the time comes. It's condition is ignored. This is useful if an action is to perform operations in parallel, then it can be described as a rule.

The Truck Example Revisited

An enhancement to the truck rule set would be to have the truck turn and move forwards at the same time. This could be done by using parallel actions in certain rules.e.g. We can combine rules 6 and 7:

Rule	*Condition*	*Action*
1	unloaded & at depot	finished
2	loaded & at depot	unload
3	loaded & facing depot	go forwards
4	loaded	turn left
5	next to bin	load bin
6		approach bin

Where the action "approach bin" would be a rule that contains two *rule sets* as parallel actions:

```
rule action: turn_towards; parallel; action: move_towards.
```

One rule set (*turn_towards*) keeps the truck facing the bin:

Rule	*Condition*	*Action*
1.1	bin to left or directly behind	turn left
1.2	bin to right	turn right
1.3		do nothing

The other rule set (*move_towards*) moves the truck in the direction of the bin:

Rule	*Condition*	*Action*
2.1	bin in front within 45 degrees left or right	move forwards
2.2		do nothing

Once the truck has reached the bin, rule 5 kicks in which stops rule 6. Stopping rule 6 stops the two movement rule sets which in turn stop their currently running rules. Only when rule 6 has stopped completely will rule 5's action be started.

Another enhancement would be to reuse the new rule sets to move towards the depot as well as the bin. This can be done by specifying that the new rule sets move towards not the truck but, say, *target*. Target can be set in the precepts at first to be the bin and later after load bin is executed to be the depot. This will give us a main rule set looking something like:

Rule	*Condition*	*Action*
1	unloaded & at depot	finished
2	loaded & at depot	unload
3	loaded	target=depot, approach target
4	next to bin	load bin
5		target=bin, approach target

Caveats and Notes

There are a few caveats to consider when using the TrPonder system. These are described here. More will be added based on users' experiences.

Use of Non-Percepts Values in Conditions

If a rule condition is not based on a value in percepts, but rather it references some other external managed object's state or value, the programme writer has to ensure that those conditions will get a chance to be re-evaluated. For example let us assume the following T-R programme:

Rule set 1		
Rule	*Condition*	*Action*
1	a > 4	myaction
2		ruleset 2

Rule set 2		
Rule	*Condition*	*Action*
1	c > 4	a++
2		c++

In TrPonder this would be written as:

Toggle line numbers

```
1 percepts at: "c" put: 1.
2 a := 1;
3
```

```
ruleset2 add: (rule condition: [ c > 4 ] action: [ a := a + 1 ]).
ruleset2 add: (rule condition: [ true ] action: [ percepts at: "c" put: (c + 1) ]).

ruleset1 add: (rule condition: [ a > 4 ] action: myaction).
ruleset1 add: (rule condition: [ true ] action: ruleset2).

percepts tell: ruleset1.
ruleset1 run: percepts.
```

Once the *c* variable has reached 5, the T-R programme will start increasing the variable *a*. However, since *a* is not in the percepts, no events are fired to signal the change in its value. Therefore, even when *a* becomes 5, the top-most rule will not get fired. This situation is the result of the following two semantic details: 1) conditions are re-evaluated when a percept value changes. 2) conditions in the currently running rule-set are re-evaluated after a rule has finished. Since *a* is not in percepts an event is not received and since the condition ($a > 4$) is not in the current running rule set. The mentioned condition does not get a chance to be re-evaluated. The easiest way around this problem is to constrain the conditions to be based only on percepts.

Rules Are Not Re-entrant

Rules are not (yet) themselves thread safe so any one instance of a rule can only be used at any one time. Do not give the same rule to different rule sets.

PonderTalk Blocks

There are a few problems inherent in the way that PonderTalk's blocks work. A block is a closure, that is it contains a copy of its environment at the time the block is created and so it is difficult to maintain common information between blocks. This is where the usefulness of the percepts comes in. The copy of the environment is a shallow copy which means that only the values of the top level environment variables are copied not the objects that they refer to. Thus if a variable refers to a domain, then then new copy of the variable will refer to the same domain, in this way shared values can be facilitated.

Toggle line numbers

```
// Create a new hash
var := #() asHash.
var at: "size" put: 10.

b1 := [ ... var at: "size" put: 12. ... var := 7. ..].
b2 := [ ... root print: (var at: "size"). ... ].

b2 value. // prints 10
b1 value.
b2 value. // prints 12

b1 value. // error number does not understand "at:put:"
b2 value. // prints 12

var := 4.
b2 value. // prints 12

```

After the blocks are created, the value of *var* is copied into each block but they still both refer to the same PonderTalk Hash managed object.

Download TrPonder

Note: This is a work in progress and may be changed at any time. The current files are dated 11th October 2009.

The TrPonder src zip file can be downloaded here and the TrPonder binary Jar file is here. The source file should be unzipped inside your Ponder2 installation `src` directory. It will create some files in `net/ponder2` and some files in `resource`. To run the example use the build.xml file you already have in the Ponder2 installation:

```
ant run -Dboot=trmaze.p2
```

The binary file can simply be added to your classpath in the build.xml file and run as above.

P2Android

Contents

This is an extension to Ponder2 that allows you to run Ponder2 on the Android phones. There are new Managed Objects that map straight onto the Android widgets on the screen and also to some of the Android sensors like the GPS location, magnetic compass and motion sensors.

The PonderDoc documentation for the managed objects can be found here.

Other P2Android information can be found in the following pages

- Introduction
- Getting started
- Simple Example

Downloads

Download the P2Android Java library to get started

P2Android (last edited 2010-08-30 09:26:43 by KevinTwidle)

P2AndroidIntroduction

Ponder2 Android Introduction

A set of classes have been written for the Android phone to map the screen elements to Ponder2 ManagedObjects. The screen layouts are created in the usual way with elements to be used by Ponder2 given unique names. Ponder2 managed objects are included in a library that support the different types of GUI widgets. They are created in Ponder2 by giving them the name of the screen widget which ties them together.

This implementation is not complete and should only be used for trial systems. Due to limitations in the way Ponder2 handles ManagedObjects certain actions taken by the Android OS cannot be recovered from. Ponder2 currently has no standard way of saving the state of managed objects so if the application is thrown out of memory it cannot recover when restarted. It is an aim of Ponder2 to be able to migrate managed objects and when this is possible they will also be able to be saved and restored within the Android system

P2AndroidIntroduction (last edited 2011-04-28 21:29:10 by KevinTwidle)

Ponder2Downloads

Ponder2 Downloads

Ponder2 is made available under the terms of the GNU Lesser General Public License as published by the Free Software Foundation.

Ponder2 Download Archive Files

These files comprise the sources, documentation and runnable binaries for any operating system. Java 1.5 is the only requirement. The version number is derived from the Subversion repository version number and does not indicate the actual number of releases.

NB. Ponder2 requires the ANTLR runtime for the PonderTalk compiler and the tuProlog runtime for the authorisation policies. From release 2.809 onwards the runtime libraries are included in the distribution along with their licences, they can be found in the `lib` directory. The Ant build files in previous releases download the libraries from their host sites when necessary. From release 2.3586 the XmlBlasterClient library and licence files are also included.

Version 2.3623

ChangeLog 22nd February 2011

Files	*Description*	*Extracts to*

ponder2-2.3623.zip	An archive with the main Ponder2 and Ponder2 Communications jar files and all the API and PonderTalk module documentation. This is all that is required to run a Ponder2 SMC without sources. An example source tree and ANT `build.xml` file are included for user-written managed objects.	`ponder2/lib`
		`ponder2/src`
		`ponder2/doc`

ponder2src-2.3623.zip	An archive with all the Ponder2 sources and ant build files. Run `ant install` to build and create everything that is included in the binary distribution.	`ponder2/p2src`
ponder2authexample-2.3623.zip	Ponder2 Authorisation example files. See Ponder2Authorisation for more details.	`ponder2/example`

Version 2.3619

ChangeLog

Files	*Description*	*Extracts to*

ponder2-2.3619.zip	An archive with the main Ponder2 and Ponder2 Communications jar files and all the API and PonderTalk module documentation. This is all that is required to run a Ponder2 SMC without sources. An example source tree and ANT `build.xml` file are included for user-written managed objects.	`ponder2/lib`
		`ponder2/src`
		`ponder2/doc`

ponder2src-2.3619.zip	An archive with all the Ponder2 sources and ant build files. Run `ant install` to build and create everything that is included in the binary distribution.	`ponder2/p2src`
ponder2authexample-2.3619.zip	Ponder2 Authorisation example files. See Ponder2Authorisation for more details.	`ponder2/example`

Version 2.3617

ChangeLog

Files	*Description*	*Extracts to*

ponder2-2.3617.zip	An archive with the main Ponder2 and Ponder2 Communications jar files and all the API and PonderTalk module documentation. This is all that is required to run a Ponder2 SMC without sources. An example source tree and ANT `build.xml` file are included for user-written managed objects.	`ponder2/lib`
		`ponder2/src`
		`ponder2/doc`

ponder2src-2.3617.zip	An archive with all the Ponder2 sources and ant build files. Run `ant install` to build and create everything that is included in the binary distribution.	`ponder2/p2src`
ponder2authexample-2.3617.zip	Ponder2 Authorisation example files. See Ponder2Authorisation for more details.	`ponder2/example`

Version 2.3593

ChangeLog

Files	*Description*	*Extracts to*

ponder2-2.3593.zip	An archive with the main Ponder2 and Ponder2 Communications jar files and all the API and PonderTalk module documentation. This is all that is required to run a Ponder2 SMC without sources. An example source tree and ANT `build.xml` file are included for user-written managed objects.	`ponder2/lib`
		`ponder2/src`
		`ponder2/doc`

ponder2src-2.3593.zip	An archive with all the Ponder2 sources and ant build files. Run `ant install` to build and create everything that is included in the binary distribution.	`ponder2/p2src`
ponder2authexample-2.3593.zip	Ponder2 Authorisation example files. See Ponder2Authorisation for more details.	`ponder2/example`

Version 2.3586

(The large version number jump is a consequence of reorganising the SVN repository)
ChangeLog

Files	*Description*	*Extracts to*

ponder2-2.3586.zip	An archive with the main Ponder2 and Ponder2 Communications jar files and all the API and PonderTalk module documentation. This is all that is required to run a Ponder2 SMC without sources. An example source tree and ANT `build.xml` file are included for user-written managed objects.	`ponder2/lib`
		`ponder2/src`
		`ponder2/doc`

ponder2src-2.3586.zip	An archive with all the Ponder2 sources and ant build files. Run `ant install` to build and create everything that is included in the binary distribution.	`ponder2/p2src`
ponder2authexample-2.3586.zip	Ponder2 Authorisation example files. See Ponder2Authorisation for more details.	`ponder2/example`

Version 2.897

ChangeLog

Files	*Description*	*Extracts to*

ponder2-2.897.zip	An archive with the main Ponder2 and Ponder2 Communications jar files and all the API and PonderTalk module documentation. This is all that is required to run a Ponder2 SMC without sources. An example source tree and	`ponder2/lib`
		`ponder2/src`
		`ponder2/doc`

	ANT `build.xml` file are included for user-written managed objects.	
ponder2src-2.897.zip	An archive with all the Ponder2 sources and ant build files. Run `ant install` to build and create everything that is included in the binary distribution.	`ponder2/p2src`
ponder2authexample-2.897.zip	Ponder2 Authorisation example files. See Ponder2Authorisation for more details.	`ponder2/example`

Version 2.840

ChangeLog

Files	*Description*	*Extracts to*

ponder2-2.840.zip	An archive with the main Ponder2 and Ponder2 Communications jar files and all the API and PonderTalk module documentation. This is all that is required to run a Ponder2 SMC without sources. An example source tree and ANT `build.xml` file are included for user-written managed objects.	`ponder2/lib`
		`ponder2/src`
		`ponder2/doc`

ponder2src-2.840.zip	An archive with all the Ponder2 sources and ant build files. Run `ant install` to build and create everything that is included in the binary distribution.	`ponder2/p2src`
ponder2authexample-2.840.zip	Ponder2 Authorisation example files. See Ponder2Authorisation for more details.	`ponder2/example`

Version 2.834

ChangeLog

Files	*Description*	*Extracts to*

ponder2-2.834.zip	An archive with the main Ponder2 and Ponder2 Communications jar files and all the API and PonderTalk module documentation. This is all that is required to run a Ponder2 SMC without sources. An example source tree and ANT `build.xml` file are included for user-written managed objects.	`ponder2/lib`
		`ponder2/src`
		`ponder2/doc`

ponder2src-2.834.zip	An archive with all the Ponder2 sources and ant build files. Run `ant install` to build and create everything that is included in the binary distribution.	`ponder2/p2src`
ponder2authexample-2.834.zip	Ponder2 Authorisation example files. See Ponder2Authorisation for more details.	`ponder2/example`

Version 2.809

ChangeLog

Files	*Description*	*Extracts to*

ponder2-2.809.zip	An archive with the main Ponder2 and Ponder2 Communications jar files and all the API and PonderTalk module documentation. This is all that is required to run a Ponder2 SMC without sources. An example source tree and ANT `build.xml` file are included for user-written managed objects.	`ponder2/lib`
		`ponder2/src`
		`ponder2/doc`

ponder2src-2.809.zip	An archive with all the Ponder2 sources and ant build files. Run `ant install` to build and create everything that is included in the binary distribution.	`ponder2/p2src`

ponder2authexample-2.809.zip	Ponder2 Authorisation example files. See Ponder2Authorisation for more details.	`ponder2/example`

Version 2.783

ChangeLog

Files	*Description*	*Extracts to*
ponder2-2.783.zip	An archive with the main Ponder2 and Ponder2 Communications jar files and all the API and PonderTalk module documentation. This is all that is required to run a Ponder2 SMC without sources. An example ANT `build.xml` file is included	`ponder2/lib`
ponder2src-2.783.zip	An archive with all the Ponder2 sources and ant build files. Run `ant build` to compile everything. Run `ant install` to create the libraries in `ponder2/lib`	`ponder2/src`
ponder2authexample-2.783.zip	Ponder2 Authorisation example files. See Ponder2Authorisation for more details.	`ponder2/example`

Version 2.731

ChangeLog

Files	*Description*	*Extracts to*
ponder2-2.731.zip	An archive with the main Ponder2 and Ponder2 Communications jar files and all the API and PonderTalk module documentation. This is all that is required to run a Ponder2 SMC without sources. An example ANT `build.xml` file is included	`ponder2/lib`
ponder2src-2.731.zip	An archive with all the Ponder2 sources and ant build files. Run `ant build` to compile everything. Run `ant install` to create the libraries in `ponder2/lib`	`ponder2/src`
ponder2authexample-2.731.zip	Ponder2 Authorisation example files. See Ponder2Authorisation for more details.	`ponder2/example`

Version 2.709

ChangeLog

Files	*Description*	*Extracts to*
ponder2-2.709.zip	An archive with the main Ponder2 and Ponder2 Communications jar files and all the API and PonderTalk module documentation. This is all that is required to run a Ponder2 SMC without sources. An example ANT `build.xml` file is included	`ponder2/lib`
ponder2src-2.709.zip	An archive with all the Ponder2 sources and ant build files. Run `ant build` to compile everything. Run `ant install` to create the libraries in `ponder2/lib`	`ponder2/src`
ponder2authexample-2.709.zip	Ponder2 Authorisation example files. See Ponder2Authorisation for more details.	`ponder2/example`

Version 2.669

ChangeLog

Files	*Description*	*Extracts to*
ponder2-2.669.zip	An archive with the main Ponder2 and Ponder2 Communications jar files and all the API and PonderTalk module documentation. This is all that is required to run a Ponder2 SMC without sources.	`ponder2/lib`

	An example ANT `build.xml` file is included	
ponder2src-2.669.zip	An archive with all the Ponder2 sources and ant build files. Run `ant build` to compile everything. Run `ant install` to create the libraries in `ponder2/lib`	`ponder2/src`
ponder2authexample-2.669.zip	Ponder2 Authorisation example files. See Ponder2Authorisation for more details.	`ponder2/example`

Version 2.432

ChangeLog

Files	*Description*	*Extracts to*
ponder2-2.432.zip	An archive with the main Ponder2 and Ponder2 communications jar files. This is all that is required to run a Ponder2 SMC	`ponder2/lib`
ponder2comms-2.432.zip	Contains library files required to use Ponder2 communications with Web Services.	`ponder2/commslib`
ponder2doc-2.432.zip	Ponder2 API documentation.	`ponder2/doc`
ponder2src-2.432.zip	An archive with all the Ponder2 sources and ant build files. Requires *ponder2comms.zip* to compile Ponder2 communications	`ponder2/src`

Version 2.427

ChangeLog

Files	*Description*	*Extracts to*
ponder2-2.427.zip	An archive with the main Ponder2 and Ponder2 communications jar files. This is all that is required to run a Ponder2 SMC	`ponder2/lib`
ponder2comms-2.427.zip	Contains library files required to use Ponder2 communications with Web Services.	`ponder2/commslib`
ponder2doc-2.427.zip	Ponder2 API documentation.	`ponder2/doc`
ponder2src-2.427.zip	An archive with all the Ponder2 sources and ant build files. Requires *ponder2comms.zip* to compile Ponder2 communications	`ponder2/src`

Version 2.423

ChangeLog

Files	*Description*	*Extracts to*
ponder2-2.423.zip	An archive with the main Ponder2 and Ponder2 communications jar files. This is all that is required to run a Ponder2 SMC	`ponder2/lib`
ponder2comms-2.423.zip	Contains library files required to use Ponder2 communications with Web Services.	`ponder2/commslib`
ponder2doc-2.423.zip	Ponder2 API documentation.	`ponder2/doc`
ponder2src-2.423.zip	An archive with all the Ponder2 sources and ant build files. Requires *ponder2comms.zip* to compile Ponder2 communications	`ponder2/src`

Ponder2 Source Repository

The various parts of the project are held in a Subversion repository at the Ponder2 repository. *This* may *only be available to members of Imperial College.*

The sub directories are:

Ponder2	The core code, documentation and jar files

Ponder2Comms	The remote inter-Ponder2 communications package
Ponder2Example	The source of the introductory example
Ponder2Tutorial	A tutorial given at Imperial College

CategoryPonder2Installation

ChangeLog

Change Log

This page details all the changes made to the Ponder2 software. Changes may be made at any time and this page will let you know whether the change is large or small and help you decide whether you need the latest version. Changes are marked by the Subversion version number of the latest commit of the main program in `ponder2.jar`.

The version number is the number printed out by the shell when you connect to the SMC using telnet.

Version 2.xxxx - Next version to be released

These are changes made to the core system that have not been released yet. They will appear in the next release.

- None yet

Known Problems in current release

- Shell reports old version number - reports the SVN number of when it was last checked in

Version 2.3623 - Latest released version

Date: 22nd February 2011

Category: Minor bug fix, enhancement

Bugfix

- Fixed authorization policy documentation (`p_source` is actually `p_subject`)
- Fixed comment that gave an error in Javadoc

Enhancement

- Added == operator for all Managed Objects.
- Added -Dauth=[allow|deny] to Ant run script

Version 2.3619

Date: 15th February 2011

Category: Minor enhancement

Enhancement

- Added global variables `p_source` and `p_target` to the condition blocks of authorisation policies.

Version 2.3617 - Latest released version

Date: 29th November 2010

Category: Minor Bugfix and enhancements

Bugfix

- Sending a Null object remotely caused an error

Enhancements

- Improved Timer managed object for one-shot and repetitive actions and events.
- Single Socket Communications protocol included for use with mobile phones..

Version 2.3593

Date: 3rd May 2010

Category: Major Missing file from release

Bugfix

- Event Managed Object's PonderTalk adaptor was missing from the Ponder2 Jar file

Features

- New comms protocol plug-in added. Single Socket Communications (SSC) allows bi-directional, concurrent messaging over a single socket. The new comms library is not released yet. Email me for more information and for the library itself. Email is address on front page.

Version 2.3586

Date: 23rd March 2010

Category: Major new features, many bug fixes

Features

- Internal improvements to the PonderTalk compiler including better error messages
- **include "filename"** includes file *filename* into the PonderTalk source file. The file must be located on the classpath or as an absolute file or relative to current location.
- New *Semaphore* managed object for multi-threading applications
- SelfManagedCell (root): added *address*: message to create local addresses dynamically and to load protocols dynamically from PonderTalk
- References to hashes are sent remotely now, not the hash itself. If a local copy is required then use **localHash := (remoteHash asArray) asHash.** because arrays are always copied remotely.
- Events, a subclass of Hash, are immutable and are copied remotely.
- New Ponder2 Communications protocols added for intra-SMC communications
    - XmlBlaster protocol
    - SSComms a new, lightweight, communications protocol utilisating a single socket for bidirectional, asynchronous traffic. Ideal for mobile phones which can only create IP connections.

Bugfixes

- Wrong number of arguments caught at compile time. Indicated by generating self-explanatory bad Java code. Not possible to report it as a compiler error.
- Empty string was sent remotely incorrectly, ended up as null.
- Empty block "[]" not allowed, use "[true]" instead.
- Certain words used by the PonderTalk grammar were treated as possible input by the lexical analyser, no longer true.
- Remote address now sent with remote objects to remote SMCs.
- If a received object that has a local address cannot be found locally then a remote error is generated.
- Fixed a problem with blocks assuming that their arguments were always local objects
- Objects that have fixed instances e.g. True and False, could be imported incorrectly if there were more than one in a message.

Version 2.897

Date: 12th September 2009

Category: Minor new features, no bug fixes

- New PonderTalk accessible arguments can be given to the SMC when it is started
- New function added to Array
    - **at:** anIndex **put:** anObject
- Collections - Domain and Hash now have

    - **size**
    - **has:** aName
    - **hasObject:** anObject
    - **remove:** anObject - to remove *anObject*
    - **removeObject:** anObject - to remove all occurences of *anObject*
- Collections - Array now has
    - **size**
    - **has:** anIndex
    - **hasObject:** anObject
    - **remove:** anIndex
    - **removeObject:** anObject - to remove all occurences of *anObject*
- Block
    - **valueVars:** aHash - to supplement/override the enclosing variables
- Numbers and Strings have better comparisons for **hasObject:** and **removeObject**
- Internal object references have better comparisons for the above operations.
- Removed Security manager from RMI protocol. Use `-Djava.security.manager=` instead
- Random commands added for numbers and booleans
- Java APT factory for **@Ponder2op** can be built without requiring an existing previous version

Version 2.840

Date: 28th April 2009

Category: Major

- Added missing NOT operation for Boolean - use as `bool not.`
- Removed possible race condition when an Event is sent to different threads

Category: Minor

- Added experimental onError: message to Block - see P2Block. This may change in the future.
- Added Ponder2 Error object, handed as an argument to the error block - see P2Error
- Improved documentation for basic objects

Version 2.834

Date: 21st November 2008

Category: Major

- Added focus "st" to authorisation policies
- Fixed authorisation checking to allow messages to objects with no parents e.g. "Hello, " + "World!".

Category: Minor

- Ant build includes all `lib` Jars so that user defined libraries may be added easily
- Ant build includes Ponder2 documentation .css file in `ponder2.jar` - affects Ponder2 documentation layout

Version 2.809 - Latest released version

Date: 4th September 2008

Category: Major

- Added java.policy security file for RMI
- Ant build file automatically checks and starts the `rmiregistry` if necessary
- Added user source tree with example and ant build file

Category: Minor

- Simplified complete source build

- Added ANTLR and tuProlog runtime libraries to distribution
- Various small improvements to basic objects
- Added ability to send remote PonderTalk via RMI, see PonderTalk managed object
- Now accepts commas between file names within a single `-boot` argument

Version 2.783

Date: 14th July 2008
Bastille Day

Category: Major

- Fixed problem when using RMI to some non-local hosts

Category: Minor

- Better error reporting when an error occurs within a block
- Various small improvements to basic objects
- Improved error handling when path name could not be resolved
- Allow all Ponder2 exceptions to hold source file information
- Added EventForwarder to control event propagation between SMCs
- Added error block to blocks
- Added error block to obligation policies
- Added `whileTrue:` and `whileFalse:` to blocks
- Added ping SMC command, returns true if remote responds
- Added sleep SMC command
- Added Missions and Interfaces
- Allow "_" character in pathnames
- To/From XML now manages recursive structures properly
- Block method now returns a block not an adaptor
- Java nio package no longer used - not available on all small devices
- Added "has: anObjectName" operation to Domain
- RMI communications tries reconnecting on error
- Socket protocol always uses new sockets to cope with overlapping requests
- Web Service protocol rewritten

Version 2.731

Date: 21st February 2008

Category: Major

- **build.xml: Javadoc - added classpath for jars of non source items.**
- JavaDoc: Fixed javadoc warnings
- PonderDoc: Added missing comments
- auth system: Provided for alternative authorisation system.
- proximity event bus: policies can now be `attach:`ed to any managed object

Version 2.709

Date: 13th February 2008

Category: Major

- Add Web Services protocol module
- Added divide (/) to numbers
- Added '!' character into the parser set of permissible binary operation characters
- Refactored underlying authorisation code

- Removed compilation dependencies of some Managed Objects upon their Adaptor Objects
- New authorisation examples
- Improved distribution scripts
- Added style sheets to PonderDoc documentation
- Rewritten ANTLR grammar file, distinguishing between path names and identifiers properly
- ANTLR grammar produces slightly more efficient abstract syntax tree

Version 2.669

Date: 11th January 2008

Category: Major

- Added floating point numbers in multiple formats and hex e.g. 3.0 3.4e6 0x0ff23
- Added authorisations with examples
- Added PonderTalk string execution
- Checks for Null in P2String
- Null Pointer Error fixed when getting to EOF
- Read PonderTalk from a file using PonderTalk Managed Object
- Shell: ls -l shows object class rather than external managed object
- Allows first element of pathname to be a variable
- Fixed problem in parser. "a + b msg" was parsed incorrectly.
- Changed collate: to collect: throughout.
- New PonderTalk/XML RMI interface
- Removed return type from PonderTalk do: operation
- Completely reorganised internal class hierarchy. All objects are managed objects. The managed object code is lazily evaluated so that there is minimal overhead.

Version 2.432

Date: 3rd July 2007

Category: Minor

- **Improved Ponder'Talk Managed Object**
- Fixed problem where """ characters were not handled properly in strings
- Improved error messages. Stack traces printed only when tracing on

Version 2.427

Date: 27th June 2007

Category: Major

- Fix to parsing of arrays. Variables/names were not included properly
- Added and: *and* or: *to Boolean*
- Strings now allow escaped " characters i.e. \"
- Fixed JAR recipes to include generated files properly

Version 2.423

Date: 25th June 2007

Category: Major

- First release of Ponder2 version 2 incorporating PonderTalk
- PonderTalk Compiler and interpreter
- Easier to write Java objects
- Improved shell

Version 1.287

Date: 30th Mar 2007

Category: Minor

- Fixed incorrect name of library in Ponder2 build.xml
- The line was changed to `<property name="qdparser" location="${dist}/qdparser.jar"/>`

Version 1.283

Date: 29th Mar 2007

Category: Major

- Added authorisation policies and examples

Note: This release was previously released incorrectly as version 1.1 containing the wrong source files. This version should be downloaded to use authorisation policies.

Version 161

Date: 12th Dec 2006

Category: Minor

- Added revision number and date to shell and main routine
- Added `-version` option to print out the current version number and then exit

ChangeLog (last edited 2011-02-22 14:56:09 by KevinTwidle)