YAWL - Technical Manual

Version 5

Contents

Document Control

Arthur ter Hofstede	version 1.9	September 2008	Consolidation of ten previous documents, conversion to LaTeX of nine of them, general cleaning and extensions.
Stephan Clemens	version 1.91	September 2008	Input for installation section.
Michael Adams	version 2.1	August 2010	Major rewrite and upgrade for 2.1. First public release of manual.
Michael Adams	version 2.2	August 2011	Enhancement and upgrade for 2.2.
Michael Adams	version 4.3	February 2020	Updates for YAWL version 4.
Michael Adams	version 5	December 2022	Updates YAWL version 5.

Original Sources

The first version of this document (1.9) combined the following earlier documents:

1. A "Connecting Custom Services to the YAWL Engine" (Beta 7 Release) document by Marlon Dumas, Tore Fjellheim and Lachlan Aldred. This document has been entirely replaced by Chapter 3, written by Michael Adams.
2. A "YAWL Core Classes Cheat Sheet - A Short Architectural Introduction" (Beta 7 Release) by Lachlan Aldred. This forms the basis for Chapter 2.
3. A "Implementing In-Process Custom Services" document created by Andrew Hastie, subsequently reviewed and refined by Lachlan Aldred, and later refined and updated by Michael Adams. This forms the basis for Chapter 4.
4. A "YAWL - Time Service Manual" (Beta 7 Release) document first created by Sean Kneipp with later updates by Tore Fjellheim and Lachlan Aldred. The Time Service has since been deprecated and so this document has been removed from the manual.
5. A document on Interface B first created by Michael Adams with a later addition by Tore Fjellheim. Parts of the document have been folded into Chapter 3.
6. A document on Interface D created by Guy Redding. Interface D has since been deprecated and so this document has been removed from the manual.
7. A document on Interface X created by Michael Adams. This forms the basis for Section 3.5.3 of Chapter 3.

8. A "YAWLEditor Technical Manual" created by Lindsay Bradford. This document was modified by Michael Adams and became Chapter 7.

9. A "newYAWL Resource Perspective Requirements Analysis" document created by Lindsay Bradford. This document was copied into Appendix A.

Chapter 1

Introduction

This chapter provides a brief background introduction to YAWL and the YAWL Foundation.

1.1 What is YAWL?

Based on a rigourous analysis of existing workflow management systems and workflow languages, a new workflow language called YAWL (Yet Another Workflow Language) was developed by Wil van der Aalst (Eindhoven University of Technology, the Netherlands) and Arthur ter Hofstede (Queensland University of Technology, Australia) in 2002. This language was based on the one hand on Petri nets, a well-established concurrency theory with a graphical representation, and on the other hand on the well-known Workflow Patterns (www.workflowpatterns.com). The Workflow Patterns form a generally accepted benchmark for the suitability of a process specification language. Petri nets can capture quite a few of the identified control-flow patterns, but they lack support for the multiple instance patterns, the cancellation patterns and the generalised OR-join. YAWL therefore extends Petri nets with dedicated constructs to deal with these patterns.

YAWL offers the following distinctive features:

- YAWL offers comprehensive support for the control-flow patterns. It is the most powerful process specification language for capturing control-flow dependencies.
- The data perspective in YAWL is captured through the use of XML Schema, XPath and XQuery.
- YAWL offers comprehensive support for the resource patterns. It is the most powerful process specification language for capturing resourcing requirements.
- YAWL has a proper formal foundation. This makes its specifications unambiguous and automated verification becomes possible (YAWL offers two distinct approaches to verification, one based on Reset nets, the other based on transition invariants through the WofYAWL editor plug-in).
- YAWL has been developed independent from any commercial interests. It simply aims to be the most powerful language for process specification.
- For its expressiveness, YAWL offers relatively few constructs (compare this e.g. to BPMN!).
- YAWL offers unique support for exceptional handling, both those that were and those that were not anticipated at design time.
- YAWL offers unique support for dynamic workflow through the Worklets approach. Workflows can thus evolve over time to meet new and changing requirements.

- YAWL aims to be straightforward to deploy. It offers a number of automatic installers and an intuitive graphical design environment.
- YAWL's architecture is Service-oriented and hence one can replace existing components with one's own or extend the environment with newly developed components.
- The YAWL environments supports the automated generation of forms. This is particularly useful for rapid prototyping purposes.
- Tasks in YAWL can be mapped to human participants, Web Services, external applications or to Java classes.
- Through the C-YAWL approach a theory has been developed for the configuration of YAWL models. For more information on process configuration visit www.processconfiguration.com.
- Simulation support is offered through a link with the ProM (www.processmining.org) environment. Through this environment it is also possible to conduct post-execution analysis of YAWL processes (e.g. in order to identify bottlenecks).

1.2 Obtaining the Latest Version of YAWL

As new versions of the YAWL Environment are released to the public, they will be available for download at the YAWL website (https://yawlfoundation.github.io) or directly from the Github repository (https://github.com/yawlfoundation/yawl). From this site it is also possible to access and/or clone the source code of all components for development purposes.

1.3 The YAWL Foundation

For up-to-the-minute information on any aspect of the YAWL Initiative, visit the YAWL Foundation Homepage (yawlfoundation.org). The YAWL Foundation is a non-profit organisation that acts as custodian of all intellectual property (IP) related to YAWL and its support environment.

1.4 Documentation

Apart from this technicial manual, there is a user manual on YAWL and a number of case studies. These studies provide detailed examples that you may wish to consult in order to obtain a deeper understanding of the application of YAWL.

This manual does not really cover the control-flow *concepts* of YAWL in detail. One reason for this is that there are quite a few papers out there that do provide this information. We refer the reader to e.g. [4] for a justification of the extensions of Petri nets introduced for YAWL on the basis of the original control-flow patterns. The main paper on YAWL, from a language point of view, is [5]. In this paper you find a formalisation of the control-flow concepts of YAWL. More recently, a CPN formalisation of newYAWL (control-flow, data and resource perspectives) was presented in [16]. For a formalisation of the OR-join, a complex synchronisation concept in YAWL, we refer to [20]. This definition supersedes the definition provided in [5].

As mentioned above, YAWL extends Petri nets. There are a number of general introductions to Petri nets in the literature. We refer the interested reader to [12, 11].

Wil van der Aalst has written much about the application of Petri nets to workflow, see e.g. [1]. The subclass of Petri nets introduced by him, Workflow-nets, is a predecessor of YAWL. The textbook that he wrote together with Kees van Hee is highly recommended reading [3].

A recent textbook on Business Process Management (BPM), which covers the original control-flow patterns and also YAWL, was written by Mathias Weske [18]. This textbook also covers other approaches, such as the modelling standard BPMN (note that the BPMN2YAWL tool can convert these specifications to YAWL).

On the YAWL web site (yawlfoundation.github.io) it can be seen how the original control-flow patterns can be realised in YAWL (follow the link on Resources and then click 'patterns'). For control-flow patterns in newYAWL the reader can consult appendix A.1 of Nick Russell's PhD thesis [16].

If you would like to know more about how verification of YAWL specifications really works, we refer you to [17] and to [19]. This work forms the theoretical basis of how the verification mechanisms are realised in the YAWL editor.

In-depth discussion of YAWL's exception handling framework from a conceptual point of view can be found in [16, 14] and from an implementation aspect in [6, 7]. YAWL's worklet approach to dealing with on-the-fly changes to workflows is discussed in [6, 8].

YAWL has a close link to the Process Mining environment ProM [2], www.processmining.org. This link is for example exploited in [13] to provide simulation support for YAWL. There exists support for exporting YAWL logs to ProM which can subsequently be analysed by one of the many mining plug-ins available in this environment.

Alternative ways of presenting work lists have been addressed in [9]. In this *Visualiser* framework users can choose a map (not just a geographical map, but also e.g. a timeline or a YAWL specification) and work items can be positioned on this map and be shown in a colour that reflects their level of urgency (a context-specific notion which can be defined for the user). Support for the visualiser framework is provided via the default YAWL worklist handler.

Finally, a textbook on YAWL, called *Modern Business Process Automation: YAWL and its Support Environment* has been published by Springer (2010; ISBN: 978-3-642-03120-5) [10]. The book provides a comprehensive treatment of the field of Business Process Management (BPM) with a focus on Business Process Automation. It achieves this by covering a wide range of topics, both introductory and advanced, illustrated through and grounded in the YAWL language and support environment.

Chapter 2

The Main Engine Classes

This chapter is a very brief overview of the main YAWL engine classes and packages, which is designed to sufficiently orient the reader so that he/she can get started. It is not intended to be comprehensive, but as a starting point from which further exploring can take place.

This chapter is intended for developers who are directly interacting with the core engine classes and packages. The reader is assumed to already have some proficiency in defining a business process specification in YAWL.

2.1 Elements Package `[org.yawlfoundation.yawl.elements]`

This package contains the elements of a yawl process model, i.e. those things that a process modeller thinks about when designing a process, such as the Atomic Tasks, Composite Tasks etc. The most important classes are:

YSpecification: objects of this class are the outer container of a process model. Such an object is a process specification/model. A process specification in YAWL basically contains a set of YNets (process nets).

YDecomposition: an abstract class sub-classed by YNet and YAWLServiceGateway. These classes are the equivalent of a net and task decomposition respectively.

YNet: an object of this class contains a set of inter-connected tasks (action elements) and conditions (stateful elements). In addition to the parameter definitions inherited from its superclass (YDecomposition) YNet allows local variables to be defined.

YTask: This abstract class is the superclass of the two types of concrete task (YAtomicTask & YCompositeTask). This class contains possibly the lion's share of the business process execution logic. It also contains many internal state elements (such as YInternalCondition).

YAtomicTask: an atomic task represents an atomic unit of work (from the perspective of the workflow engine).

YCompositeTask: a composite task represents a task that decomposes down to a sub-net (i.e. another YNet object).

YAWLServiceGateway: in order for an atomic task to actually do something it needs to decompose to a YAWLServiceGateway. Think of this a special kind of decomposition for an atomic task. It is basically used to define the input parameters and output parameters of the task, and it can optionally define which YAWL Custom Service must be invoked when its task is enabled.

YCondition: a condition is like a place in Petri nets. It is a sibling member to the tasks inside any net. They typically sit between tasks, and store the process state 'identifiers' ('tokens' in Petri speak). When two tasks are directly connected together in a YAWL process model (in the XML syntax), an "invisible" condition is created between the tasks, when the process is loaded into the engine.

YFlow: objects of this class represent the arcs that join tasks to conditions in a process model. They refer to both their preceding task/condition and to their succeeding task/condition. Likewise each task/condition is "aware" of the flows preceding it, and the flows succeeding it. Therefore tasks/conditions, and their flows form up a bi-directional linked list, of sorts (except it's more of a graph than a list, but you get the idea).

2.2 State Package `[org.yawlfoundation.yawl.elements.state]`

This is an important package and contains classes used to store and process the state of the process control flow. Therefore, state does not refer to the variables and parameters of the process, but state as in which tasks are active, enabled etc. Important classes include:

YIdentifier: objects of this class represent the things that flow through the YNets indicating its process state. If you are familiar with Petri nets then consider the references to objects of this class the tokens. Due to the fact that composite tasks contain nets of their own the identifiers are capable of creating children. The children pass through the subordinate nets. This idea is fully described in the YAWL Book [10].

2.3 Engine Package `[org.yawlfoundation.yawl.engine]`

This is the package responsible for running the process and progressing it through its elements as per its control flow. Important classes of this package are:

YEngine: this singleton is responsible for storing all of the active process specifications in object format. It is also a single control point for all operations over running process instances, e.g. launching cases, starting work-items (check-out), completing work-items (check-in), cancelling cases etc. It delegates some of these process instance controlling operations to some YNetRunner (see below) objects, however the engine stores and aggregates each YNetRunner instance and correlates it with the YIdentifier object running through it.

YNetRunner: Executes its YNet instance. Each YNet instance is essentially scheduled and controlled by YNetRunner. When an XML process specification gets loaded into the engine some instances of YNet get created. Collectively, these YNet instances form the process template. When a process instance (case) is launched, each YNet instance gets deep-cloned by the engine, and the copy is then wrapped by the YNetRunner, which executes it.

YWorkItem: objects of this class are created when a task is enabled. Their identification consists of the net based YIdentifier and the Task identifier. When an enabled task is checked out a new workitem is spawned off from the enabled workitem. It is not just a matter of changing state from 'enabled' to 'fired' due to the fact that some tasks are multi-instance. Therefore all 'fired' work items are children of their previously enabled parents.

YWorkItemRepository: is a store of work-item instances for the engine.

YNetRunnerRepository: is a store of YNetRunner instances for the engine.

Chapter 3

YAWL Custom Services: Creation and Deployment

During the execution of a YAWL specification instance, the YAWL Workflow Enactment Engine (the *Engine*, for short) is essentially responsible for two things:

1. The correct *scheduling* of tasks, according to the control-flow defined within the specification; that is, determining the order of task execution.
2. The management of the data input into, and output from, each task, and its interplay with the task's containing net and/or the external environment.

However, the Engine is *not* responsible for the actual execution of the task. For each and every atomic task scheduled by the Engine for execution, responsibility for its execution is delegated by the Engine to a so-called *YAWL Custom Service*.

A YAWL Custom Service may be defined as a web-based service that is able to (i) receive notifications from the Engine when a task is scheduled to be delegated to it; (ii) inform the Engine that it has taken responsibility for its execution; (iii) perform the task's activities, as appropriate; and then (iv) inform the Engine that the task's execution has completed, allowing the Engine to continue processing the specification instance and determine the next task(s) for scheduling (if the instance has not completed). This service-oriented, delegated execution framework is the extensibility cornerstone of the YAWL System.

At design time, each and every atomic task in a specification is associated with a Custom Service. The association is either explicitly set by a designer, or else implicitly with whatever Custom Service is configured in the Engine to be the "Default Worklist Handler" (which by default is the Resource Service). At runtime, when a task is scheduled, the Engine will notify whichever Custom Service has been associated with the task that there is a task ready to be delegated to it.

This chapter describes how to create and deploy a YAWL Custom Service. Note that Custom Services are defined as services that are responsible for runtime task execution, and thus form a subset of a more general class of services and applications that may interact with the YAWL Engine, such as the YAWL Monitor Service and the YAWL Editor, both of which do *not* execute tasks. While the description that follows focusses on YAWL Custom Services, much of what is discussed is also applicable to the interactions of those more general services and applications.

3.1 Fundamentals

Custom Services interact with the YAWL Engine through XML/HTTP messages processed by certain endpoints, some located on the Engine side and others on the service side. In principle, Custom Services can

be developed in any programming language and can be deployed on any platform capable of sending and receiving HTTP messages, and may be deployed locally or remotely to the Engine. Custom Services are registered with the Engine by specifying basic authentication credentials, and a location in the form of a "base URL". Once registered, a Custom Service may receive XML messages from the Engine at endpoints identified by URLs derived from the base URL provided at registration. On the other hand, the Custom Service can send XML messages to the Engine at endpoints identified by URLs that the Custom Service is assumed to know.

A collection of Java classes included in the YAWL distribution provide several APIs that can be used to implement the required endpoints without requiring any knowledge about the URL encoding and XML formatting used. Specifically, a Custom Service will use an API known collectively as *Interface B* to define the interactions involved in taking responsibility for and performing the execution of a task's activities. Four basic interactions are particularly relevant at this stage (Figure 3.1):

1. The interaction that the engine initiates to notify that a task is scheduled for execution. Immediately prior to the notification, the Engine creates a new *work item*, using the task definition as a template, and marks its status as *enabled*. The notification message generated by the Engine contains various identifiers describing the work item, such as case id, name, specification identifiers and so on.

2. The interaction that the service initiates to inform the Engine that it is ready to perform the work item, known as a *check-out* of the work item. When this occurs, the Engine processes the work item's input data, and includes it in the data structure returned to the Custom Service. The Engine also moves the work item state from "enabled" to "executing", denoting that a Custom Service is currently executing the work item (i.e. the work item is "in progress").

3. The interaction that the Engine may initiate to notify a Custom Service that a work item in the "executing" state should be cancelled.

4. The interaction that the Custom Service initiates to indicate that it has completed execution of the work item, known as a *check-in* of the work item and its output data back to the engine, which moves the work item from "executing" to "completed" state.

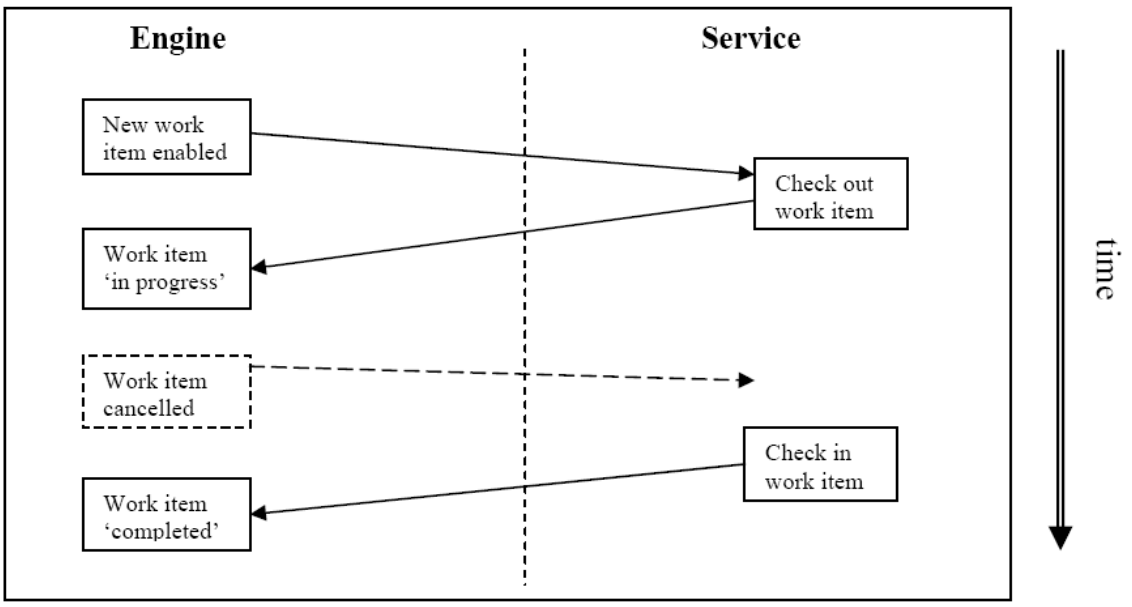


Figure 3.1: Interface B Basic Interactions

An overview of the various interfaces in the YAWL Environment is shown in Figure 3.2. This Chapter describes the interfaces between the YAWL Engine and external services (i.e. Interfaces A, B, E and X). The API interface to manage Custom Forms is discussed in Chapter 6.

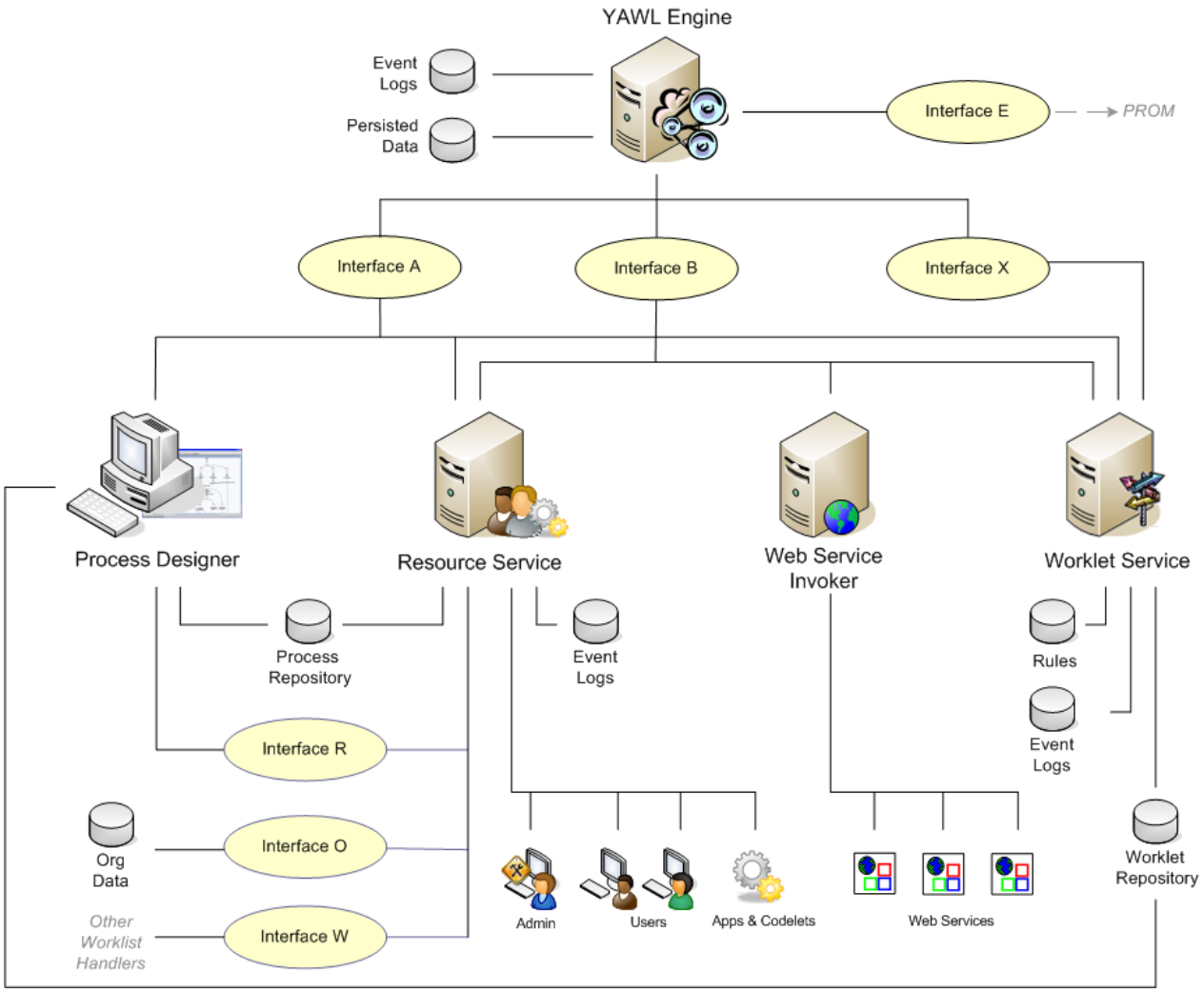


Figure 3.2: An overview of the YAWL architecture (taken from [10])

Before delving further into Custom Services, it is important to have a clear understanding of the following points[1]:

The Difference Between Atomic and Composite Tasks An atomic task represents a unit of work that typically is to be performed by an external entity, such as a person, web service or application, and is executed by delegating its activities to a Custom Service. A composite task represents a placeholder for a sub-net, and is executed by decomposing (or unfolding) into the sub-net contained within it, and scheduling the contents of the sub-net as per its control-flow. A composite task is never delegated to a Custom Service.

The Difference Between a Task and a Work Item Amongst other elements, a YAWL process specification contains a number of atomic tasks which describe a unit of work to be performed as a part of the overall process. That is, each atomic task is a *definition* of the work to be performed, and includes definitions of its

[1]For the inquiring reader, the concepts discussed here and throughout this chapter are covered in much greater detail in [10]

control-flow, data and resourcing aspects. At runtime, each atomic task definition acts as a template for the creation of one or more *work items*. That is, a work item is a runtime instance derived from an atomic task definition. An analogy might be: an atomic task is to a work item as a blueprint is to a house constructed from that blueprint. The lifetime of a task is equal to the lifetime of its containing process specification. The lifetime of a work item always falls within its containing process instance.

Task Decompositions A task decomposition is essentially a 'contract' between an atomic task and the external environment. That is, each atomic task that is required to interact with an external entity (external to the Engine, such as a person, application or service) must reference a decomposition. Amongst other things, a decomposition is responsible for the definition of task-level input and output variables, and for specifying which Custom Service will be responsible for the task's execution at runtime. Several atomic tasks in a process may reference the same decomposition. A task without a decomposition is a so-called *empty* task and is handled entirely within the Engine itself.

Work Item Creation When the Engine determines that an atomic task is ready to be scheduled, it creates a work item derived from the task definition, sets the work item's status to 'Enabled', and announces the work item's enablement to the Custom Service specified in the task's decomposition. Along with the announcement, an XML representation of the work item is passed to the Custom Service. An enabled work item does *not* contain instantiated data values for its input variables, since data mappings only take place immediately before the work item begins execution.

Work Item Check-out When a Custom Service indicates to the Engine that it is ready to check-out a work item, the Engine takes the enabled work item and *creates a clone of it*. The status of the original work item is changed to 'IsParent', while the cloned work item (known as a *child* work item) is given the status 'Executing' and has its data values instantiated (the reasons for this mechanism will become clearer in the discussion of multiple-instance tasks later in this chapter). An XML representation of the *child* work item, including instantiated data values for its input variables, is returned to the Custom Service as the result of a successful check-out.

Work Item Identifiers Each work item created by the Engine is given a unique identifier, which consists of two parts, a case index and a task identifier, separated by a colon (:). The task identifier also consists of two parts, the task's name and a numeric string suffix assigned by the engine to ensure task identifier uniqueness, separated by an underscore (_). Thus, an example work item id for a task named 'Enrol' might be `123:Enrol_5`. The case index value of an enabled work item equates to the case identifier of its containing net (thus '123' in the example also refers to the case id of the root (or only) net of the process). When a task is set to 'Executing' status, a unique digit is added to its case index, using a dotted notation. So, for example, a parent work item with an identifier of `123:Enrol_5` may have a executing child with identifier `123.1:Enrol_5`, to denote the first child work item created. Consequently, the work item passed to a Custom Service with a notification of enablement, and the work item returned to a Custom Service as the result of a successful check-out, ***refer to two different work items***.

A Note on Resourcing While the Engine is responsible for the control-flow and data perspectives of a process instance, it has no involvement in the resource perspective. That is, the Engine has no concept of how a work item is to be resourced, but delegates responsibility for that to a Custom Service (for example, the YAWL Resource Service). Therefore, concepts such as offering and allocation of work items do not exist within the Engine. Rather, the Engine merely schedules (enables) work items, and delegates their execution to Custom Services. How a Custom Service handles the delegation is entirely up to the Custom Service involved; for example, the Resource Service adds the necessary "semantic layers" on top of an enabled work item to denote it as 'offered' or 'allocated', but other Custom Services have no need for or knowledge of these added semantics.

With those points covered, the next section describes the creation of a simple Custom Service.

3.2 A Simple Custom Service

This section describes the steps to follow to create a basic Custom Service. The YAWL Twitter Service will be used as a running example. The Twitter Service, which may be used to send a status update to Twitter, is an example of a very simple Custom Service; the entire service consists of a single class compiled from about 140 lines of Java code.

At the core of each Custom Service are the following Interface B classes:

- **org.yawlfoundation.yawl.engine.interfce.interfaceB.InterfaceB_EnvironmentBasedServer**
  This class is a Java servlet and is responsible for receiving event notifications from the Engine.

- **org.yawlfoundation.yawl.engine.interfce.interfaceB.InterfaceB_EnvironmentBasedClient**
  This class provides methods that allow a Custom Service to call specific endpoints on the engine side.

- **org.yawlfoundation.yawl.engine.interfce.interfaceB.InterfaceBWebsideController**
  An abstract utility class that encapsulates much of the functionality of the other two classes, and is designed to be extended by the primary class of each Custom Service.

These classes (and their dependent classes, as explained later) may be sourced from the YAWL engine distribution (.war file), or any existing Custom Service distribution, or the *yawl-lib* jar, or the YAWL Standalone jar file.

3.2.1 Creating the Service's Primary Class

The first step is to create a new class that extends from `InterfaceBWebSideController` – this will be the main class (and in the case of the Twitter Service, the only class) of the new Custom Service. Besides a number of helper methods and data members that the Custom Service inherits, `InterfaceBWebSideController` contains two abstract methods that all Custom Services must implement:

- *public void handleEnabledWorkItemEvent(WorkItemRecord workItemRecord)* This method is the endpoint of an Engine enabled work item event notification; that is, it is called when the Engine creates a new work item and places it in the enabled state, and the Custom Service containing the method is the one associated at design time with the work item.

- *public void handleCancelledWorkItemEvent(WorkItemRecord workItemRecord)* This method is invoked when an enabled or executing work item is cancelled by the engine (e.g. due to a cancellation occurring within the process control-flow). In principle, after this method has been invoked, a Custom Service should not attempt to check-out or check-in the work item, and if the work item is in progress, its processing should be stopped[2].

Note that only the Custom Service associated with a particular work item will receive notifications via these two methods.

Notice that both methods pass a `WorkItemRecord` object to the Custom Service. As noted earlier, the Engine side of the API passes an XML String representation of a work item together with these event notifications; `InterfaceBWebSideController` unmarshals the XML String, using its values to construct an `org.yawlfoundation.yawl.engine.interfce.WorkItemRecord` object. `WorkItemRecord` is a convenience class designed to make it easier to interrogate the work item descriptors received without having to parse the XML itself. Table 3.1 contains the descriptors of a `WorkItemRecord` object; each has corresponding *get* and *set* methods, and each is stored as a String value, except for 'datalist', which is a JDOM Element. These members will be discussed as they arise later in this chapter.

[2]Note that YAWL does not support any notion of "atomic execution", so no "rollback" is required when a work item is cancelled.

Table 3.1: Data members of a `WorkItemRecord`

Data Member	Description
allowsDynamicCreation	'true' if dynamic work item creation is supported (multiple instance tasks only)
caseID	case index of the work item
codelet	codelet to invoke for this work item
completedBy	name of the service that checked the work item in
completionTime	date/time stamp of the work item's completion time
completionTimeMs	work item's completion time in milliseconds (as a String)
customFormURL	URL of the custom form to display for this work item
dataList	original data list Element (executing work items only)
dataListString	the data list as an XML String
dataListUpdated	updated data list element (executing work items only)
dataListUpdatedString	updated data list as an XML String
edited	'true' if data has been updated
enablementTime	date/time stamp of the work item's enabled time
enablementTimeMs	work item's enabled time in milliseconds (as a String)
extendedAttributes	work item's task-level extended attributes
firingTime	date/time stamp of the work item's firing time
firingTimeMs	work item's firing time in milliseconds (as a String)
logPredicateCompleted	configurable log value for work item's completion
logPredicateStarted	configurable log value for work item's starting
requiresManualResourcing	'true' if work item requires resourcing
resourceStatus	current resourcing status of the work item
specIdentifier	the unique key value of the specification that contains this work item
specURI	the name of the specification that contains this work item
specVersion	the version of the specification that contains this work item
startedBy	name of the service that checked the work item out
startTime	date/time stamp of the work item's start time
startTimeMs	work item's start time in milliseconds (as a String)
status	current work item status
tag	user-defined value
taskID	the work item's task identifier
taskName	the name of the task this work item was created from
timerExpiry	date/time stamp when the work item's time will expire (if any)
timerTrigger	when the work item's timer was or will be triggered
uniqueID	an added unique identifier (deprecated).

3.2.2 Implementing the handleEnabledWorkItemEvent method

A Custom Service is responsible for ensuring that each work item delegated to it is checked out of the engine, and later returned to the Engine, together with its updated data as required, when it is deemed complete. The actions that a Custom Service performs between check-out and check-in are entirely in the hands of its developer. For simpler services, the basic technique is to encapsulate this behaviour within the implementation of the *handleEnabledWorkItemEvent* method, so that it checks out the enabled work item, performs the desired action, and then checks the work item back into the engine.

Before a Custom Service can successfully check out a work item (or perform any other action via the API), it must first establish a valid session with the Engine. There are two API methods inherited from `InterfaceBWebSideController` that serve this purpose:

- *public String connect(String userID, String password)*
  Attempts to create a session with the engine by passing authentication credentials and returning a unique *session handle*.
- *public boolean checkConnection(String sessionHandle)*
  Returns true if the session handle is currently valid.

The credentials passed to the `connect` method must match those entered for the Custom Service when it is registered with the Engine (see Section 3.3.4 for details). If successful, a session handle (String) is returned that should be stored by the service for subsequent use whenever an API method is called. The session will be maintained by the Engine for 60 minutes of inactivity (by default, or as configured); that is, a session handle will expire if it is not used in an API call within 60 consecutive minutes (each API call restarts the timer). The `checkConnection` method should therefore be called immediately before each API method call to ensure the session is still active.

As an illustration of what's been discussed so far, Listing 3.1 shows the Twitter Service's entire public-facing code, including the two abstract method implementations. Note the following:

- The TwitterService class extends `InterfaceBWebsideController` (line 1).
- A global String variable to store a session handle is declared (line 4).
- Final `userid` and `password` variables are declared (lines 6 & 7). These credentials must match those registered for the service within the Engine. Note that `InterfaceBWebsideController` encrypts the password before it is sent across the API.
- The two abstract 'event' methods mentioned above are implemented (lines 9–25, and line 27). Note that in this example the cancelled work item event is implemented as an empty method for simplicity (In the unlikely event that a cancel event is received while the `handleEnabledWorkItemEvent` method is still processing, it will be handled by catching the exception that will be thrown later by the `checkinWorkItem` method). More complex Custom Services would implement this method to stop processing of the work item and take any required compensatory actions.
- The first thing the `handleEnabledWorkItemEvent` method does is to ensure it has a valid connection to the Engine. It does that by first calling its own `connected` method, and if that returns false, calls the connect method with the userid and password declared above, which if successful will assign a valid session handle value to the `_handle` variable. Again for simplicity, the handle is not checked again after the call to `connect`, since an invalid session handle will be caught by the `checkOut` method throwing an exception.
- The `connected` method calls the `checkConnection` method only if the session handle is not currently `null` (lines 43–45).
- Again for simplicity, the `catch` block (lines 21–23) handles any Exception generically; it is recommended, of course, that each of the exceptions that may the thrown in the `try` block is handled in a more appropriate way.

```
1  public class TwitterService extends InterfaceBWebsideController {
2
3      // holds a session handle to the engine
4      private String _handle = null;
5
6      private final String _engineUser = "twitterService";
7      private final String _enginePassword = "yTwitter";
8
9      public void handleEnabledWorkItemEvent(WorkItemRecord wir) {
10         try {
11
12             // connect only if not already connected
13             if (! connected()) _handle = connect(_engineUser, _enginePassword);
14
15             // checkout ... process ... checkin
16             wir = checkOut(wir.getID(), _handle);
17             String result = updateStatus(wir);
18             checkInWorkItem(wir.getID(), wir.getDataList(),
19                             getOutputData(wir.getTaskID(), result), null, _handle);
20         }
21         catch (Exception ioe) {
22             ioe.printStackTrace();
23         }
24     }
25
26     // have to implement abstract method, but have no need for this event
27     public void handleCancelledWorkItemEvent(WorkItemRecord workItemRecord) {  }
28
29     // these parameters are automatically inserted (in the Editor) into a task
30     // decomposition when this service is selected from the list
31     public YParameter[] describeRequiredParams() {
32         YParameter[] params = new YParameter[2];
33         params[0] = new YParameter(null, YParameter._INPUT_PARAM_TYPE);
34         params[0].setDataTypeAndName("string", "status", XSD_NAMESPACE);
35         params[0].setDocumentation("The status message to post to Twitter");
36
37         params[1] = new YParameter(null, YParameter._OUTPUT_PARAM_TYPE);
38         params[1].setDataTypeAndName("string", "result", XSD_NAMESPACE);
39         params[1].setDocumentation("The status result or error message returned");
40         return params;
41     }
42
43     private boolean connected() throws IOException {
44         return _handle != null && checkConnection(_handle);
45     }
```

Listing 3.1: The Twitter Service's public methods

After the connection has been established and/or checked, the method *checkOut* is invoked to check out the enabled work item. This method requires two parameters: the enabled work item's ID, which is retrieved by using the `WorkItemRecord#getID()` method, and a valid session handle (see Listing 3.1, line 16). If successful, the *checkOut* method returns a `WorkItemRecord` representing the executing, child work item. Remember, the `WorkItemRecord` returned by the `checkout` method represents a different work item instance from the one passed to the `handleEnabledWorkItemEvent` method:

- their identifiers are different.
- the checked-out work item has 'Executing' status (instead of 'Enabled').
- the checked-out work item has an instantiated `datalist`.

ASIDE: The TwitterService, like most simpler Custom Services, reuses the same `WorkItemRecord` object to store the checked-out work item, effectively discarding the original, since it has no further need to reference the initial, enabled work item. More complex Custom Services may need to keep the original work item as well, in which case the distinction between them becomes very important.

Once the work item has been checked-out, the Custom Service can perform the appropriate actions for it. In our example, an `updateStatus` method is called (line 17 of Listing 3.1), which takes the data contained in the work item and uses it to connect to Twitter, pass it the specified status update and retrieve the resulting message (the actual machinations of how this is achieved is not of concern here).

When the appropriate action has been completed, the work item can be checked back into the Engine by calling the `checkInWorkItem` method. The method has five parameters:

- *String itemid*: the identifier of the work item being checked in. Note that this must be the identifier of the checked-out work item, and not that of the work item passed to the `handleEnabledWorkItemEvent` method.
- *Element inputData*: the set of input data variables and values as contained in the checked-out work item. In almost all cases, a call to the `WorkItemRecord`'s `getDataList` method is used to fulfil this parameter.
- *Element outputData*: the set of output data variables and values required (see below for more).
- *String logPredicate*: a piece of configured text to be logged in the Engine's process logs along with other logged values representing this work item's completion. A `null` argument is acceptable if there is no log predicate.
- *String handle*: a valid session handle.

The `outputData` parameter requires a correctly formatted JDOM Element containing a complete list of output data variables and values expected by the Engine for this work item. Our TwitterService example uses a `getOutputData` method to construct an appropriate Element; that method can be seen in Listing 3.2.

Note that the `getOutputData` method receives two parameters: the name of the task (retrieved by calling the checked-out `WorkItemRecord#getTaskID` method on line 19 of Listing 3.1) and the result returned from the Twitter Service. The `getOutputData` method creates a containing Element with the same name as the task (line 2); it is a mandatory requirement that the output data Element has the same name as the task represented by the work item. Next, it creates another Element called "result" — this is the name of only output variable for work items handled by the Twitter Service — and assigns the value contained in the variable `data` to it. Finally, it adds the "result" Element (representing the output variable and its value) as a child to the `taskName` Element (representing the container) and returns the constructed Element to be passed back to the Engine via the `checkInWorkItem` method. An example of how the Element would appear as XML can be seen in Listing 3.3.

ASIDE: In our TwitterService example, we have assumed that each API method call is successful and thus returns the expected result. In reality, if an API method does not succeed, it may throw an exception or

```
1     private Element getOutputData(String taskName, String data) {
2         Element output = new Element(taskName);
3         Element result = new Element("result");
4         result.setText(data);
5         output.addContent(result);
6         return output;
7     }
```

Listing 3.2: The Twitter Service's getOutputData method

```
1     <myUpdateStatusTask>
2         <result>Update successful.</result>
3     </myUpdateStatusTask>
```

Listing 3.3: Example contents of an Element produced by the Twitter Service's getOutputData method

return a diagnostic message describing the problem encountered. Methods that return a String value (for example, the `connect` and `checkInWorkItem` methods) should have their returned value checked instead of simply assuming the call was successful. A convenience method called `successful` is inherited from `InterfaceBWebsideController` by Custom Services that, when passed the result of an API call that returns a String value, will return a boolean `false` if the result contains a diagnostic method rather than the expected result. An example on the use of the `successful` method can be see in in Listing 3.4. For API methods that return objects to Custom Services, an exception is thrown containing a diagnostic message if the call is unsuccessful. For example, the `checkOut` method throws a `YAWLException` if the check-out was unsuccessful. Additionally, each API method call will throw an `IOException` if the Engine can't be reached.

3.2.3 Implementing the handleCancelledWorkItemEvent method

Sometimes, a work item that is being handled by a Custom Service is cancelled by the Engine before the service has checked it out or before it has completed processing it. Cancellation may occur when a work item is a member of another task's cancellation set, or when it is a member of a deferred choice construct, or when an administrator cancels a case, or as a result of an exception handling routine. In any event, when a work item is cancelled the Engine notifies the appropriate Custom Service by calling its `handleCancelledWorkItemEvent` method. The Custom Service can then take whatever action is appropriate by implementing the method (as one of two `InterfaceBWebsideController`'s abstract methods, it must be implemented by each Custom Service).

For example, the Twitter Service takes no action at all, implementing the method with an empty body. On the other hand, the Worklet Service, which launches whole case instances for the work items it handles, needs to take appropriate action to cancel any cases it may have launched on behalf of the cancelled work item; the Resource Service must take action to remove the work item from any work queues it resides in, clean up its internal caches and so on.

3.2.4 Overriding the describeRequiredParams method

A well behaved Custom Service, which expects to receive particular data variables in order to perform the actions of a work item, will override the `describeRequiredParams` method. The purpose of this method is to allow Custom Services to specify exactly what input and output variables it requires. At design time, when a designer chooses the Custom Service to associate with a task in the Editor, the Editor will call the service's `describeRequiredParams` method (using the Interface B API) and populate the task with the specified data variables, saving the designer the effort of manually creating the required variables and thereby greatly reducing the margin for errors occurring at runtime.

```
1     public void handleEnabledWorkItemEvent(WorkItemRecord wir) {
2         try {
3
4             // connect only if not already connected
5             if (! connected()) {
6                 _handle = connect(_engineUser, _enginePassword);
7
8                 if (successful(_handle) {
9                    wir = checkOut(wir.getID(), _handle);
10                   // and so on...
11                }
12                else {
13                    _log.error(_handle);
14                }
15            }
16        }
```

Listing 3.4: The successful method example

Our TwitterService example overrides the `describeRequiredParams` method to specify one input variable (status) and one output variable (result) that it expects all work items that are passed to it for handling will contain (Listing 3.1, lines 31–41). The method returns an array of `YParameter` objects, each one representing a data variable. In line 32 of our example, the method declares an array of size 2 (since there are two variables to specify). For each variable:

- It first constructs a `YParameter` object, specifying the desired usage type (which must be either `YParameter._INPUT_PARAM_TYPE` or `YParameter._OUTPUT_PARAM_TYPE`). If an *Input & Output* variable is required, two parameters must be created, one input and one output, with the same name and datatype; they will be merged at runtime. The first argument of the `YParameter` constructor refers to the variable's containing task, which has not yet been assigned, so a `null` is passed.
- Next, the variable's datatype, name and namespace are set. Unless the variable has a user-defined data type with its own namespace, the default `XSD_NAMESPACE` should be used for the namespace value.
- Next, the variable's documentation is set (optional).

The effects of the `describeRequiredParams` method can be seen in the example in Figure 3.3. It is only necessary for a developer to choose an appropriate Custom Service for the task from the drop-down list of services, its required variables then populate the list automatically.

3.3 Deployment

Our simple Custom Service class is complete, and so now needs to be bundled inside a Web Application (war) file and loaded into a servlet container (for example, Apache Tomcat) for execution. A number of other files and required libraries will also need to be placed inside the war file, in the correct structure. This section will walk through the deployment process.

3.3.1 Internal Structure of a WAR File

The general internal structure of a war file for deployment to Tomcat is briefly described below. For more detailed information, please refer to Chapter 3 of the Tomcat Application Developer's Guide[3].

[3] https://tomcat.apache.org/tomcat-9.0-doc/index.html

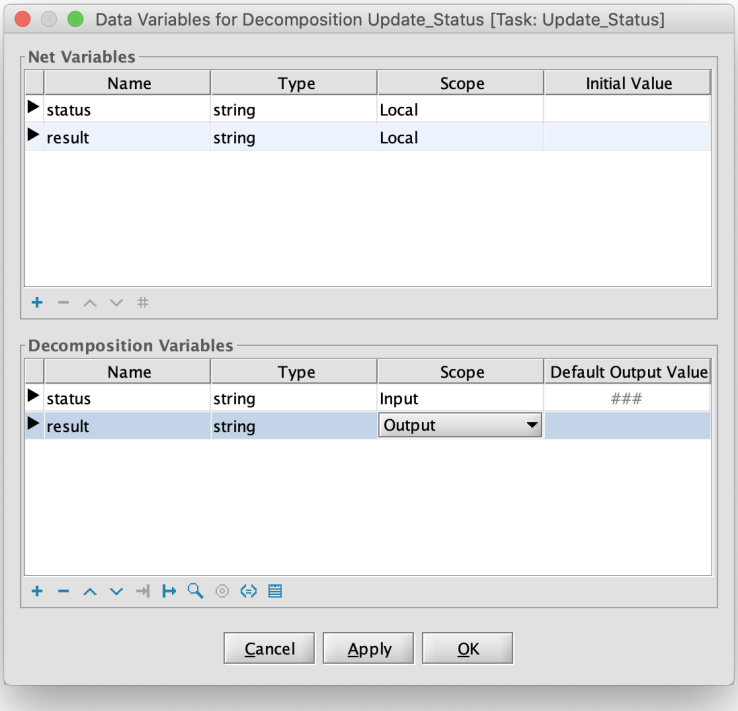


Figure 3.3: Required Variables Populated from the describeRequiredParams method

A Web Application file is a compressed file that is unpacked by Tomcat, when it is placed in the Tomcat *webapps* directory, into a sub-directory structure based on its internal organisation. Every war file will, at a minimum, conform internally to the following basic structure:

- a root directory, which typically contains any html and jsp files (and supporting files like JavaScript and style sheet files) used by the service, although more complex services may have these files arranged in sub-directories.
- a sub-directory of the root called `WEB-INF`, which contains:
  - a `web.xml` file, more formally referred to as a *Web Application Deployment Descriptor*. This file is an XML file containing configuration information for the application, and is discussed in more detail below.
  - a sub-directory called `classes`, where any required class files, that are not contained in jar libraries, are placed. The directory structure under the `classes` directory must match the package structure of any classes placed here. For example, any YAWL classes should be placed in a sub-directory structure beginning with `/org/yawlfoundation/yawl`.
  - a sub-directory called `lib`, where any jar libraries that the application requires are to be placed.

3.3.2 Required Files

This section describes the absolute minimum set of files (classes and jars) required to be bundled with a basic Custom Service. Of course, particular Custom Services will require other files, depending on what the service actually does and how it is realised. These class files can be found in any of the installed YAWL

Custom Services, or within the `yawl-lib-xx.jar`[4] library (which, if placed in the classpath of a service, negates the need to explicitly include these files in the service).

Firstly, these YAWL Interface B client side classes are mandatory inclusions for all Custom Services (package `org.yawlfoundation.yawl.engine.interfce.interfaceB`):

- `IBControllerCache`: caches recently accessed engine objects on the client side (specifications, task information, work items, and so on).
- `InterfaceB_EnvironmentBasedClient`: the service side API for service to engine calls.
- `InterfaceB_EnvironmentBasedServer`: the service side receiver of event notifications from the engine, which are passed on to the Custom Service's primary class.
- `InterfaceBWebsideController`: the base class of the Custom Service's primary class.

The above classes require the inclusion of these dependent classes:

- Package `org.yawlfoundation.yawl.elements`:
  - `YAttributeMap`: a map of attributes (key=value pairs).
  - `YSpecVersion`: stores the version descriptors for a YAWL specification.
  - `YVerifiable`: a Java interface implemented by several required classes; provides for self-verification of a class's data structures.
- Package `org.yawlfoundation.yawl.elements.data`:
  - `YParameter`: stores variable types and descriptors.
  - `YVariable`: a variable of a net or task, and the base class of YParameter.
- Package `org.yawlfoundation.yawl.engine`:
  - `YSpecification`: the container for a YAWL specification.
  - `YSpecificationID`: a unique identifier for each specification.
- Package `org.yawlfoundation.yawl.engine.interfce`:
  - `Interface_Client`: the base class of all YAWL API's; provides the basis for HTTP communication between the Engine and Custom Services.
  - `Marshaller`: converts XML Strings to their equivalent YAWL objects and vice versa.
  - `ServletUtils`: some generic servlet utility methods.
  - `SpecificationData`: stores specification definition data and descriptors on behalf of services.
  - `TaskInformation`: stores task definition descriptors, including task parameters, on behalf of services.
  - `WorkItemRecord`: stores information describing a work item on behalf of services.
  - `YParametersSchema`: stores descriptors of task-level parameter sets.
- Package `org.yawlfoundation.yawl.engine.logging`:
  - `YLogDataItem`: definition of a item to be added to the logging of a process event.
  - `YLogDataItemList`: a list of the above items.
- Package `org.yawlfoundation.yawl.util`:

[4] where *xx* refers to the current YAWL release version (e.g. yawl-lib-2.2.jar).

- `CharSetFilter`: ensures all strings passed via HTTP communications between the Engine and services use UTF-8 encoding.
- `JDOMUtil`: various utility methods for converting Strings to JDOM objects and vice versa.
- `PasswordEncryptor`: hashes passwords on the service side for passing to the engine.
- `StringUtil`: various String manipulation methods.

In addition, the above classes require these external jar libraries to be available to the Custom Service, either in the Service's `/WEB-INF/lib` directory, or elsewhere on the classpath:

- commons-codec-1.4.jar
- jdom.jar
- log4j.jar
- xercesImpl.jar

These jar files can be copied from the `/WEB-INF/lib` directory of any of the default custom services. Finally, the file `log4j.properties` needs to be copied to the `/WEB-INF/classes` directory from the same location in any of the default services.

Of course, war files representing any but the most basic of Custom Services can (and often do) include other directories, class and jar files, and other contents besides those stated. A good idea is to browse the structures of the various YAWL services to gain some insight into how things are typically arranged.

3.3.3 The Deployment Descriptor File (web.xml)

Every Custom Service must include a `web.xml` file in its `WEB-INF` directory. The file contains the necessary deployment configuration values for the service. As an example, the Twitter Service's `web.xml` is shown in Listing 3.5. This example may be copied for use by basic Custom Services - only a few values need to be changed in most cases, as described below.

In the basic web.xml file example:

- The name the Custom Service (line 4) and a brief description (line 5) are provided. These values are used only by the Tomcat Manager tool (YAWL doesn't refer to them). The values should be set to match those of your Custom Service.
- A context parameter named *InterfaceBWebsideController* is defined (lines 7–12). The value of this parameter must match the fully qualified name of your service's primary class (which extends from `InterfaceBWebsideController` class). At startup, an instance of the primary class referred to is created.
- A second context parameter named *InterfaceB_Backend* is defined (lines 14–17). The value of this parameter must match the URL of the Engine's Interface B API. For a locally installed Engine, this parameter value can be left unchanged; if the Engine is installed remotely, the URL's host name should be changed to that of the remote Engine. The standard tomcat port (`:8080`) should be changed only if the Engine's Tomcat container has modified its port from the default. In all cases, the URL must end in `/yawl/ib` – the path of the Engine's Interface B endpoint.
- A servlet character set filter and mapping is provided (lines 19–31), which should be included as-is in all Custom Service web.xml files. The filter ensures that all Strings passed between the Engine and Services will be encoded with the UTF-8 character set.
- The definition of the servlet for the Custom Service is at lines 33–39. The `servlet-class` value should not be changed - it defines the fully qualified name of the class that receives notifications from the Engine (and is inherited in your service's primary class, as it extends from `InterfaceBWebsideController`). The `servlet-name` value may be changed to match that of your service.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<web-app version="2.4">
    <display-name>TwitterService</display-name>
    <description>A twitter service</description>

    <context-param>
        <param-name>InterfaceBWebSideController</param-name>
        <param-value>
            org.yawlfoundation.yawl.twitterService.TwitterService
        </param-value>
    </context-param>

    <context-param>
        <param-name>InterfaceB_BackEnd</param-name>
        <param-value>http://localhost:8080/yawl/ib</param-value>
    </context-param>

    <filter>
        <filter-name>CharsetFilter</filter-name>
        <filter-class>org.yawlfoundation.yawl.util.CharsetFilter</filter-class>
        <init-param>
            <param-name>requestEncoding</param-name>
            <param-value>UTF-8</param-value>
        </init-param>
    </filter>

    <filter-mapping>
        <filter-name>CharsetFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <servlet>
        <servlet-name>twitterService</servlet-name>
        <servlet-class>
            org.yawlfoundation.yawl.engine.interfce.interfaceB.
                InterfaceB_EnvironmentBasedServer
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>twitterService</servlet-name>
        <url-pattern>/ib</url-pattern>
    </servlet-mapping>

</web-app>
```

Listing 3.5: The Twitter Service's web.xml file

- The definition of the servlet mapping for the Custom Service is at lines 41–44. The `servlet-name` value here and the `servlet-name` value provided in the servlet definition (in the previous point) must match. The `url-pattern` value must match the path part of the URI provided to the Engine when the Custom Service was registered; by convention, Custom Services use the path `/ib`. For example, the Twitter Service is registered with the Engine with the URI `http://localhost:8080/twitterService/ib`. The `twitterService` part matches the name of the war file deployed. The `/ib` part matches the `url-pattern` value provided here. The result is that when the Engine sends a notification the the URI it has registered for your service, that URI is mapped by Tomcat to the Servlet class defined in the previous point, allowing your Custom Service to receive those notification events.

The Twitter Service example is a minimal web.xml that suits the requirements of most Custom Services, but a web.xml can contain any number of context parameters, servlet definitions, filters and so on. For a more complex example, see the Resource Service's web.xml file.

3.3.4 Registering the Service with the Engine

The Engine requires each Custom Service wishing to communicate with it to first be registered. Registration is achieved via the *Services* administration web form (Figure 3.4), which is available to all users with administrative access. A service can be added by providing a name, a password and confirmation, a URI and a Description. The password and confirmation password must match each other, and name and password must also exactly match the credentials that will be used by the service to log onto the Engine. For example, the credentials for the Twitter Service are declared in the service's code (Listing 3.1, lines 6–7); they are used by the Service when it connects to the Engine.

The URI entered is validated by contacting it and waiting for an appropriate response, so care should be taken that (1) the Service's war file is already deployed in Tomcat, and (2) the URI provided exactly matches that specified by the service (i.e. the first part of the path matches the name of the Service's war file, and the last part matches the servlet-mapping parameter value for the `InterfaceB_EnvironmentBasedServer` servlet class as defined in the Service's web.xml – by convention the last part of the path is `/ib`).

When the *Add* button is clicked, if the credentials and URI entered are correct, the service will be successfully registered with the Engine. If not, an explanatory error message will be displayed, with more information on the problem to be found in the *localhost* and *catalina.out* log files (found in the *logs* directory of your Tomcat installation).

3.4 Additional Topics

We've now covered all the fundamentals of creating and deploying a basic YAWL Custom Service. In this section, we will examine some additional topics that extend the capabilities of Custom Services.

3.4.1 Handling Multiple-Instance Atomic Tasks

In the YAWL language, there are two kinds of atomic tasks: ordinary single-instance tasks, each of which creates one executable (child) work item, and multiple-instance tasks, each of which creates multiple executable (child) work items.

As described in Section 3.1, when a single-instance atomic task is checked out by a Custom Service, a child work item is created by the Engine, given the status 'Executing', and returned to the service. When a multiple-instance atomic task is checked out by a Custom Service, the mechanism is similar, but a little more involved:

- A (specified at design time) number of child work items are created by the Engine;
- *One* of the children, chosen indiscriminately, is given a status of 'Executing' and returned to service; and

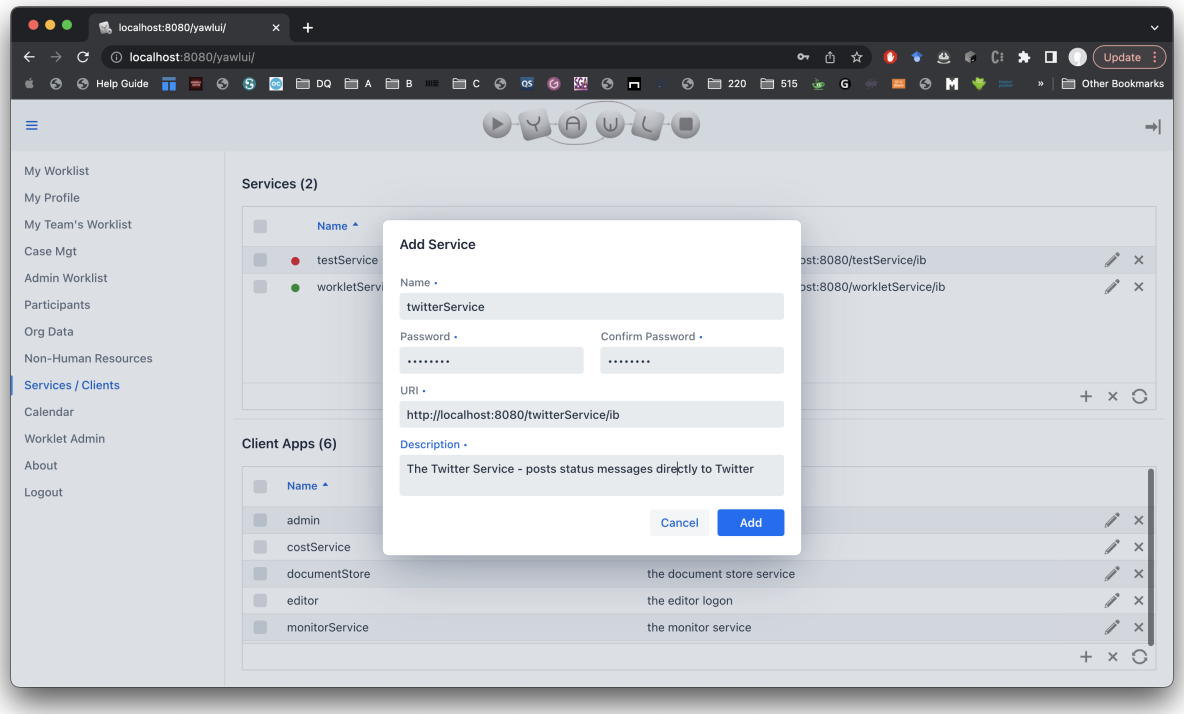


Figure 3.4: Registering a YAWL custom service

- The remainder of the child work items are given a status of 'Fired' and remain in the Engine.

A 'Fired' status indicates that the work item has progressed beyond the 'Enabled' state but is not yet 'Executing'. However, the task cannot complete until all of its child work items (or a specified threshold of those created) have been started, performed and completed by a Custom Service. It is therefore the responsibility of the Custom Service that checked out the original child work item to start, perform and complete the remaining child work items.

If your service is to handle multiple-instance tasks, there are two methods that can be used to achieve that goal:

Individual Check Outs

After the initial checkout, a Custom Service may work through the list of child work items created and check-out each of the 'Fired' items individually, as shown in Listing 3.6. Line 2 uses the inherited `getChildren` method to get a `List` of all the work items created. At this stage, one of the child items has a status of 'Executing' (the one returned by the initial `checkOut` call), the rest are 'Fired'. Note also in line 2 we use the id of the enabled work item rather than the one checked out, since the enabled item has the same id as the one that is now the *parent* of the created children (the parent work item is retained within the Engine). Each child item that has a status of 'Fired' is checked out in turn (lines 3–5) and added to the list of checked out children, except one (the one that already has 'Executing' status), which is added to the checked out list directly.

```
1   List<WorkItemRecord> checkedOut = new ArrayList<WorkItemRecord>();
2   for (WorkItemRecord child : getChildren(enabledItem.getID(), _sessionHandle)) {
3       if (child.getStatus().equals(WorkItemRecord.statusFired)) {
4           checkedOut.add(checkOut(child.getID(), _sessionHandle));
5       }
6       else checkedOut.add(child);
7   }
```

Listing 3.6: Checking out the work items of a multiple-instance task individually

```
1    public void handleEnabledWorkItemEvent(WorkItemRecord wir) {
2        try {
3
4            // connect only if not already connected
5            if (! connected()) _handle = connect(_engineUser, _enginePassword);
6
7            // checkout ... process ... checkin
8            List<WorkItemRecord> checkedOut =
9                        checkOutAllInstancesOfThisTask(wir, _handle);
10           for (WorkItemRecord child : checkedOut) {
11               String result = updateStatus(child);
12               checkInWorkItem(child.getID(), child.getDataList(),
13                        getOutputData(child.getTaskName(), result), null, _handle);
14           }
15       }
16       catch (Exception ioe) {
17           ioe.printStackTrace();
18       }
19   }
```

Listing 3.7: Checking out all the work items of a task in a single call

Encapsulated Method

For convenience, a method inherited from `InterfaceBWebsideController` is available to Custom Services that encapsulates the functionality of the code in listing 3.6. The method, `checkOutAllInstancesOfThisTask`, may be used as an alternative to the `checkOut` method and, as the name implies, takes care of checking out every child work item, including the first one. That is, there is no need to check out the original enabled item first before working through the other child items, as in the previous method. Indeed, the `checkOutAllInstancesOfThisTask` method is passed the entire enabled `WorkItemRecord` as an argument, and it will throw an Exception if its status is anything other than 'Enabled'. The method can be used for single-instance atomic tasks, too. The usage of the method is demonstrated in Listing 3.7, which shows an alternate `handleEnabledWorkItemEvent` implementation to the one in Listing 3.1.

Each work item created for a multiple-instance task may contain different data values, so each is used in a `updateStatus` call (line 10), and each is checked in individually.

A multiple instance task is not considered complete until all (or a specified threshold) of its child work items have been checked in to the Engine, which keeps track of the children checked out and back in via the internally-held parent work item from which the child items were created. Therefore, a Custom Service does not need to concern itself with the number of child work items checked out or checked in for a task, only that it eventually checks in each item that it checks out.

3.4.2 Other Interface B Events

While the two mandatory events discussed earlier must be implemented by all Custom Services, there are a number of other events generated by the Engine that a Custom Service may choose to optionally implement. They are:

handleEngineInitialisationCompletedEvent()

The Engine sends this event to all registered Custom Services immediately after it is started and completes all of its initialisation tasks. The notification of this event is useful to Custom Services because typically, since Tomcat starts web applications in alphabetical order, a Custom Service will be started before the Engine (contained in *yawl.war*). Thus, a Custom Service may use this event to undertake any final initialisation tasks that depend on the availability of a running Engine.

handleTimerExpiryEvent(WorkItemRecord item)

Any atomic task may be associated with a timer. If the timer expires before its child work items have completed, the Engine will fire this event, allowing a Custom Service that has checked out the task to perform the appropriate action.

handleCompleteCaseEvent(String caseID, String caseData)

Notifies either one or all registered Custom Services when a process instance completes. The caseData argument is an XML String of the case's output data. Some Custom Services need to be aware of case completions, for example the Worklet Service, which launches cases as substitutes for checked out work items, and checks the work items back in when the launched case completes. Whether one or all Custom Services are notified of a case completion is dependent on how the case is launched – see *Launching a Case* in the next section for more details.

handleCancelledCaseEvent(String caseID)

Notifies all registered Custom Services when a process instance is cancelled. Any Custom Service that monitors running cases will use this event to take whatever action is appropriate for the service. For example, the Resource Service uses this event to remove any work items associated with the case from its work queues, and to remove the case from the list of running cases in the *Case Management* form.

handleWorkItemStatusChangeEvent(WorkItemRecord item, String oldStatus, String newStatus)

Notifies all registered Custom Services when the status of a work item changes. Services may choose to handle this event to take any action appropriate – usually this means reacting to a status change caused by another Custom Service (since the service would already know of any changes it instigated). Note that initial statuses are not announced via this method (e.g. 'Enabled' or 'Fired'), since they are the statuses a work item begins with, a status change is not represented by a work item creation. Note also that the work item passed via this method does not contain the item's output data when a work item completes, because the output data is never incorporated into the work item, rather it is submitted separately to the Engine (via the `checkInWorkItem` method).

For examples of implementations of each of these event handlers, review those of the `ResourceManager` class of the Resource Service.

```
1     public String launchCase(YSpecificationID specID, String caseData)
2             throws IOException {
3         YLogDataItemList logData = getLogData();
4         String caseID = _interfaceBClient.launchCase(specID, caseData,
5                             _sessionHandle, logData);
6         if (! successful(caseID)) {
7             // let the user know there's a problem
8         }
9         return caseID;
10    }
```

Listing 3.8: Launching a new case

3.4.3 Launching a Case

A case (process instance) can be launched by a Custom Service via Interface B. Listing 3.8 shows a typical implementation. On line 4, the `launchCase` method of the `_interfaceBClient` object, inherited directly from `InterfaceBWebsideController`, is called with the following arguments:

- `YSpecificationID specid`: The ID of the specification from which to launch a case. There is an assumption that the specification has already been loaded into the Engine.
- `String caseData`: An XML String containing values for any net-level input variables. The root name of the XML structure must match either the name of the root net or the URI (name) of the specification, while its contents must include the correct name and valid data value for each input variable expected. If the specification has no net-level input variables, a null may be passed for this argument.
- `String sessionHandle`: A valid and active session handle previously obtained from the Engine.
- `YLogDataItemList logData`: A list of log predicate Strings to insert into the Engine's process logs when the case starts, over and above the default case launch log events. If there are no log predicates to be logged for the case launch, a null may be passed for this argument (in which case line 3 of Listing 3.8 can be ignored).

The call to `_interfaceBClient.launchCase` will return either the case ID of the newly launched case if the launch was successful, or a diagnostic error message if it was not. Line 6 of Listing 3.8 uses the inherited `successful` method to determine if the returned value represents a successful case launch – if it does not, the returned String can be used to construct an informative error message for the user.

There is a second form of the `launchCase` method, as shown in Listing 3.9, that takes one extra argument – the URI String of the Custom Service launching the case (in most cases) – known as the 'case observer'. The difference between the two methods is the effect on case completion notifications:

- For a case launched via the first method (without a case observer), *all* registered services are notified of the completion of the case;
- For a case launched via the second method (with a case observer), the observer is the *only* service notified of the completion of the case.

Note that it is acceptable for the case observer URI to differ from the URI of the Custom Service launching the case, if it is desired that a different service handles the case completion event.

```
1     String caseID = _interfaceBClient.launchCase(specID, caseData,
2                             _sessionHandle, logData, myServiceURI) ;
```

Listing 3.9: Launching a new case with a case observer

```
1     public String uploadSpecification(File specFile) throws IOException {
2         InterfaceA_EnvironmentBasedClient client =
3               new InterfaceA_EnvironmentBasedClient("http://localhost:8080/yawl/ia");
4         return client.uploadSpecification(specFile, _sessionHandle);
5     }
```

Listing 3.10: Uploading a Specification

3.5 Other Engine Interfaces

Besides Interface B, there are three other interfaces through which services and applications can interact with the YAWL Engine. This section introduces those three interfaces.

3.5.1 Interface A

The focus of Interface A is to provide support for two fundamental functions:

- The uploading and unloading of specifications; and
- Registering, modifying and removing Custom Service and Client Application accounts.

To use Interface A, a service must access the class `InterfaceA_EnvironmentBasedClient`, found in the `org.yawlfoundation.yawl.engine.interfce.interfaceA` package, together with the dependent files listed in Section 3.3.2, either within its own war file, or via its classpath. Since the Engine does not generate any event notification on Interface A, no other inclusions are necessary.

Uploading and Unloading Specifications

Uploading a specification is quite straightforward, as shown in Listing 3.10. First, an interface client is created, by passing to its constructor the URI of the Engine's Interface A endpoint (`/yawl/ia`) on lines 2–3[5], then calling the client's `uploadSpecification` method with a `File` object that references a specification file (line 4). The String returned from the method will contain an diagnostic error message if there was an error uploading the specification. These diagnostic messages can be quite informative, since the Engine verifies the syntax of the specification and reports any errors or warnings found.

A second version of the `uploadSpecification` method accepts the specification as an XML String, instead of a File object.

To unload a specification from the Engine, call the `unloadSpecification` method, passing the specification identifier and a valid session handle.

ASIDE: `InterfaceA_EnvironmentBasedClient` contains identical `connect` and `checkConnection` methods to the ones found in Interface B. Either interface's methods can be used to establish and check a connection, and the handle returned via a `connect` call on one interface can be successfully passed with the methods of the other (i.e. there is no need to have a separate session handle for each interface).

[5]Of course, the `InterfaceA_EnvironmentBasedClient` object will more typically be created outside of, and passed to, the example method shown in Listing 3.10.

Service and Application Account Maintenance

Each Custom Service and Client Application must have its credentials registered with the Engine before they can connect to it through one of its interfaces. Interface A provides the necessary methods to enable service and application registration and maintenance programmatically, rather than via the administration pages.

What makes a service a YAWL Custom Service is its ability to have the execution of a work item delegated to it. That is, every YAWL Custom Service implements Interface B and so is able to receive event notifications of work item enablements, and consequently is able to be associated with tasks at design time (in the Editor). Any web service that does not implement Interface B, and all standalone applications and tools, do not meet this criterion, but nevertheless may connect with the Engine as a Client Application, rather than as a Custom Service. For example, while the Monitor Service is a web service, deployed via a war file in Tomcat in the same way as Custom Services, it does not implement Interface B and so is regarded as a Client Application. The Editor is an example of a standalone Client Application that communicates with the Engine.

ASIDE: YAWL has always supported a generic administrative logon account called 'admin', which can be used by individuals to logon to the default administrator worklist, and by services and applications to connect to the YAWL Engine. Since the use of a generic logon may be quite insecure, from version 2.1 onwards its use has been discouraged, in favour of the unique logon credentials for each service and application as discussed in this section. While enabled by default, which allows for connections from unregistered services using the generic 'admin' account to succeed, the account may be disabled via a context-param in the Engine's `web.xml` file. Once one or more administrative level participants have been added to YAWL, and once it is determined that there are no Custom Services relying on the generic logon, you are encouraged to disable it. All of the services and applications deployed with a YAWL release use their own unique credentials.

Custom Service Registration and Maintenance

To use Interface A to register a Custom Service, the following additional classes are first required:

- `org.yawlfoundation.yawl.elements.YAWLServiceReference`: stores a reference to a Custom Service.
- `org.yawlfoundation.yawl.authentication.YClient`: the base class of the `YAWLServiceReference` and the `YExternalClient` classes.

An example of how a Custom Service may be registered with the Engine can be seen in Listing 3.11. First, an Interface A client is created (lines 3–4), then on lines 5–6 a new service reference object is constructed. The `YAWLServiceReference` constructor receives five arguments:

- `uri`: The absolute URI of the Custom Service.
- `gateway`: This argument is only relevant on the Engine side. All client side services should use `null` for this argument.
- `name`: The logon name for the service.
- `pw`: The logon password for the service. The password is supplied in plain text – it will be encrypted on the client side of the interface call.
- `doco`: A description of the Custom Service. This is the description that appears in the drop down list of services in the Editor.

The `addYAWLService` method is called on line 7, returning a diagnostic String describing the success or otherwise of the add attempt.

To remove a registered Custom Service, use the `removeYAWLService(serviceURI, sessionHandle)` method.

```
1  public String addRegisteredService(String name, String pw,
2                                     String uri, String doco) throws IOException {
3      InterfaceA_EnvironmentBasedClient client =
4          new InterfaceA_EnvironmentBasedClient("http://localhost:8080/yawl/ia");
5      YAWLServiceReference service =
6          new YAWLServiceReference(uri, null, name, pw, doco);
7      return client.addYAWLService(service, _sessionHandle);
8  }
```

Listing 3.11: Registering a Custom Service with the Engine

```
1  InterfaceA_EnvironmentBasedClient client =
2          new InterfaceA_EnvironmentBasedClient("http://localhost:8080/yawl/ia");
3
4  String msg = client.addClientAccount(name, password, doco, sessionHandle);
5
6  msg = client.updateClientAccount(name, password, doco, sessionHandle);
7
8  msg = client.removeClientAccount(name, sessionHandle);
```

Listing 3.12: Interface A operations with Client Applications

Client Application Registration and Maintenance

Client Application accounts are added, maintained and removed in a similar way to Custom Services. Listing 3.12 shows the basic methods.

When adding or updating, the password argument should be passed as plain text, as it will be encrypted by the Interface A client. When using the `updateClientAccount` method, the name argument must match that of an existing client account (i.e. the name of a client account cannot be changed). Each method returns a diagnostic String describing the success or otherwise of the call.

The generic 'admin' logon is made available as a Client Application account, allowing administrators to change its password via the `updateClientAccount` method. However, the 'admin' account cannot be removed via a `removeClientAccount` call; an attempt to do so will result in an error message being returned. To remove the account, the context parameter `AllowGenericAdminID` in the Engine's `web.xml` needs to be set to disabled.

Please consult the YAWL javadoc for more details of the complete functionality set offered by Interface A.

3.5.2 Interface E

Interface E provides access to the Engine's process logs. The are two output formats returned by the various methods of the interface, the majority are in plain XML, but case histories can also be retrieved in the OpenXES format, allowing the log output to be directly imported into the ProM process mining tool[6].

To access Interface E, the `YLogGatewayClient` class needs to be added to your Custom Service (from the package `org.yawlfoundation.yawl.engine.interfce.interfaceE`). An example of using Interface E is shown in Listing 3.13; note in particular that the Engine's Interface E URI ends with the path `/yawl/logGateway`. The boolean argument in the `getSpecificationXESLog` method, when true, includes all variable data value changes in the output returned.

There are numerous methods to retrieve logged data for such things as:

- all the events of a particular case.

[6] www.processmining.org

```
1    YLogGatewayClient client =
2            new YLogGatewayClient("http://localhost:8080/yawl/logGateway");
3
4    String xesLog = client.getSpecificationXESLog(specID, true, sessionHandle);
```

Listing 3.13: Interface E primary methods

- all the events for a particular task instance.
- all the task instances of a particular case.
- all the task instances of a task definition.
- the data type for a particular data item (complex types included).
- all the events of all cases instantiated from a particular specification.
- all the cases started and/or cancelled by a particular Custom Service.
- and many more.

Please consult the Interface E javadoc for more details.

3.5.3 Interface X

This section provides a technical overview of the structure and use of *Interface X* - an interface through which certain milestone notifications are passed during the execution of a process instance where a process exception may have occurred or may be tested for.

Interface X has been designed to allow the Engine to notify Custom Services of certain milestone events during the execution of a process instance, providing services the ability to dynamically check for, capture and handle process exceptions and/or take other actions depending on the particular milestone, process state and current data values.

An example of a service which utilises Interface X is the *Worklet Exception Service*, which has been part of YAWL distributions since the YAWL Beta 8 release. However, the use of Interface X is not limited to exception handling requirements, but is structured for 'generic application' - that is, it can be applied by a variety of services that can make use of milestone notifications during process executions.

Setup

To receive and react to event notifications via Interface X, a Custom Service needs to:

1. Implement the `InterfaceX_Service` (java) interface. An example of how to implement the required java interface can be seen in Listing 3.14.
2. Add a *servlet* section for the Interface X Service side class that receives notifications from the Engine to the Custom Service's `web.xml`. The required sections to be added can be seen in Listing 3.15 – these settings can be added as-is. This means that the Engine can find your Custom Service Interface X listening class by using the path: `/service_war_file_name/ix`. Notice the required sections are very similar to the equivalent entries for Interface B (shown in Listing 3.5).
3. Add the required Interface X service side classes to the Custom Service's war file: `InterfaceX_Service`, `InterfaceX_ServiceSideClient`, and `InterfaceX_ServiceSideServer`, all found within the package `org.yawlfoundation.yawl.engine.interfce.interfaceX`.

```
1   public class MyService implements InterfaceX_Service {
```

Listing 3.14: Implementing the Interface X Java Interface (example)

```
1   <servlet>
2       <servlet-name>InterfaceX_Servlet</servlet-name>
3       <description>
4           Listens to notification of exceptions from the engine.
5       </description>
6       <servlet-class>
7           org.yawlfoundation.yawl.engine.interfce.interfaceX.
                InterfaceX_ServiceSideServer
8       </servlet-class>
9       <load-on-startup>1</load-on-startup>
10  </servlet>
11
12  <servlet-mapping>
13      <servlet-name>InterfaceX_Servlet</servlet-name>
14      <url-pattern>/ix</url-pattern>
15  </servlet-mapping>
```

Listing 3.15: The required Interface X entries for the Custom Service's web.xml

4. Register the Custom Service with the Engine as an *InterfaceXListener*, which can be achieved in *either* of the following two ways:

   - Statically, in the *Engine's* web.xml, by adding your service's Interface X URI to the `InterfaceXListener` context-param (see Listing 3.16) – this context-param is commented out by default, so must be uncommented for use.
   - Dynamically, by calling the Interface X method `addInterfaceXListener` – typically this method is called from the Custom Service's `handleEngineInitialisationCompletedEvent` (see Listing 3.17). Notice in lines 1–2 that an `InterfaceX_ServiceSideClient` object has been instantiated, with the *Engine's* Interface X URI, in the service's initialisation code, while on line 7 the object's `addInterfaceXListener` method has been called with the *Custom Service's* Interface X URI.

The Interface X Package

The package `org.yawlfoundation.yawl.engine.interfce.interfaceX` contains the following entities:

```
1   <context-param>
2       <param-name>InterfaceXListener</param-name>
3       <param-value>http://localhost:8080/myService/ix</param-value>
4       <description>
5           This value provides the URI of an Interface X listening service.
6           Multiple uris can be specified, separated by semi-colons ';'
7       </description>
8   </context-param>
```

Listing 3.16: Adding an InterfaceXListener via the Engine's web.xml

```
1     InterfaceX_ServiceSideClient _ixClient =
2         new InterfaceX_ServiceSideClient("http://localhost:8080/yawl/ix"));
3
4     ...
5
6     public synchronized void handleEngineInitialisationCompletedEvent() {
7         _ixClient.addInterfaceXListener("http://localhost:8080/myService/ix");
8     }
```

Listing 3.17: Adding an Interface X Listener

- `ExceptionGateway`: a java interface that defines the event notifications that pass from the engine side to the custom service side.
- `InterfaceX_Service`: a java interface that defines the methods that must be implemented by the custom service to receive event notifications.
- `InterfaceX_EngineSideClient`: implements the ExceptionGateway interface. Receives the event notifications from the engine, translates them and their data sets to POST messages and passes them across the interface.
- `InterfaceX_ServiceSideServer`: a java Servlet that receives the POST messages passed across the interface from the EngineSideClient, and translates them and their data sets into method calls to the custom service (i.e. the methods implemented from the InterfaceX_Service interface).
- `InterfaceX_ServiceSideClient`: has a number of methods that are called from the custom service. Translates the method calls and their datasets into POST messages and passes them across the interface.
- `InterfaceX_EngineSideServer`: a java Servlet that receives the POST messages passed across the interface from the ServiceSideClient, and translates them and their data sets into method calls to the engine.

The logical layout of the interface can be seen schematically in Figure 3.5.

Of course, the items of interest to a Custom Service developer are those on the service side. These are described in some detail below.

InterfaceX_Service

By implementing this interface, a Custom Service must implement eight methods, seven of which are notification announcements from the engine side:

- `handleCheckCaseConstraintEvent(YSpecificationID specID, String caseID, String data, boolean preCheck)`

  This method is invoked by the Engine at two points in the lifecycle of a case, firstly when the case is launched and again when it completes. It is designed to allow a Custom Service to check the case's starting and concluding case-level data to ensure they are valid, and to take appropriate action as required, but of course may be used for other purposes.

  - `specID` is the identifier of the specification from which the case was instantiated;
  - `caseID` is the identifier of the case instance;
  - `data` is an XML String containing the current case-level variable values; and
  - `preCheck` denotes whether the case has just been launched (true), or it has just completed (false).

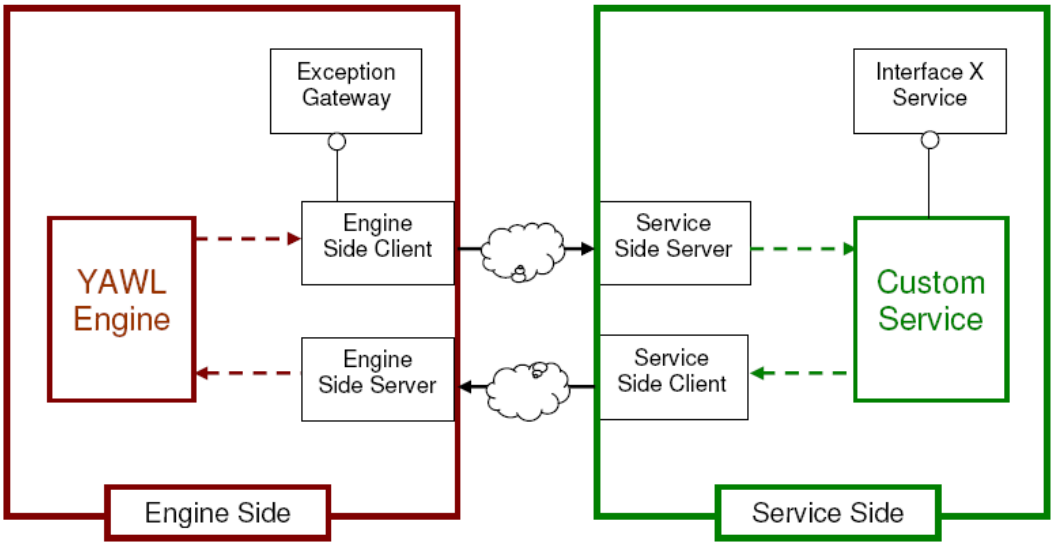


Figure 3.5: Schematic Overview

- `handleCheckWorkItemConstraintEvent(WorkItemRecord wir, String data, boolean preCheck)`

  This method is invoked by the Engine at two points in the lifecycle of a work item, firstly when the workitem is enabled and again when it completes. It is designed to allow a Custom Service to check the workitem's input and output data respectively to ensure it is valid, and to take appropriate action as required before and after the workitem is executed, but of course may be used for other purposes.

  - `wir` is the workitem record of the item in question;
  - `data` is a XML String of containing data values of the work item's variables; and
  - `preCheck` denotes whether the work item has just been enabled (true), or it has just completed (false).

- `handleTimeoutEvent(WorkItemRecord wir, String taskList)`

  This method is invoked by the Engine when a work item timer reaches its expiry deadline.

  - `wir` is the work item record that has expired; and
  - `tasklist` is a concatenated String of task identifiers of the form "[taskA, taskB, taskX]" representing a list of tasks which had work items running in parallel to the timed workitem when the timeout occurred.

- `handleCaseCancellationEvent(String caseID)`

  This method is invoked by the Engine when a case is cancelled.

  - `caseID` is the identifier of the case instance.

- `handleResourceUnavailableException(WorkItemRecord wir)`

  This method is invoked, not by the Engine, but by the Resource Service, whenever a work item is assigned to a resource and that resource is not currently available or does not exist.

- `wir` is the work item record that has an unavailable resource.

- `handleWorkItemAbortException(WorkItemRecord wir)`
  `handleConstraintViolationException(WorkItemRecord wir)`

  These two methods are included in the interface for future implementation - that is, they are not yet implemented on the Engine side and so are never called. However, since they are included in the interface, any Custom Service that implements it must include these definitions as empty methods.

- `doGet(HttpServletRequest request, HttpServletResponse response)`

  Handles the servlet request that occurs when a browser client navigates to the custom service's URI. Should be used to display a web page.

InterfaceX_ServiceSideClient

The Service-Side Client class provides a number of methods that may be called from the Custom Service to be invoked on the Engine side. The list below provides a brief overview of each. Most require an active session with the engine.

`String addInterfaceXListener(String listenerURI) throws IOException`

Purpose: Registers the URI of the Custom Service with the engine. If successful, enables the custom service to receive notification of events on Interface X.

Arguments: `listenerURI` - the full URI of the custom service (e.g. "http://localhost:8080/myCustomService/ix")

Returns: a `String` denoting success or failure.

`String removeInterfaceXListener() throws IOException`

Purpose: Removes the URI of the Custom Service from the engine, effectively ceasing Interface X event notifications.

Arguments: nil

Returns: a String denoting success or failure.

`void updateWorkItemData(WorkItemRecord wir, Element data, String sessionHandle) throws IOException`

Purpose: Maps the data values passed to the workitem specified on the engine side. Used to update the values of a workitem's data attributes, stored in the engine, with the values of identically named attributes in the Element passed.

Arguments: `wir` - the workitem to update;
`data` - a JDOM Element containing data attributes and values;
`sessionHandle` - the session handle of an active session.

Returns: nil

`void updateCaseData(String caseID, Element data, String sessionHandle) throws IOException`

Purpose: Maps the case level data values specified to the identified case on the Engine side. Used to update the values of case-level data attributes, stored in the Engine, with the values of identically named attributes in the Element passed.

Arguments: `caseID` - the identifier of the case to update
`data` - a JDOM Element containing data attributes and values
`sessionHandle` - the session handle of an active session.

Returns: nil

```
void forceCompleteWorkItem(WorkItemRecord wir, Element data, String sessionHandle)
throws IOException
```

Purpose: Force-completes the specified workitem. A force-completed workitem receives a final status of "ForcedComplete" and the process continues to the next task.

Arguments: `wir` - the workitem to force-complete;
`data` - a JDOM Element containing the workitem's final data values;
`sessionHandle` - the session handle of an active session.

Returns: nil

```
WorkItemRecord unsuspendWorkItem(String workItemID, String sessionHandle)
throws IOException
```

Purpose: Continues (i.e. unsuspends) the specified workitem. Does nothing if the workitem is not already suspended.

Arguments: `workItemID` - the identifier of the workitem to continue;
`sessionHandle` - the session handle of an active session.

Returns: A `WorkItemRecord` referring to the updated workitem.

```
void restartWorkItem(String workItemID, String sessionHandle) throws IOException
```

Purpose: Restarts the specified workitem. Sets its status as "Enabled" and resets the item's data values to those when it was initialised.

Arguments: `workItemID` - the identifier of the workitem to restart;
`sessionHandle` - the session handle of an active session.

Returns: nil

```
void startWorkItem(String workItemID, String sessionHandle) throws IOException
```

Purpose: Starts the specified workitem in the engine.

Arguments: `workItemID` - the identifier of the workitem to start;
`sessionHandle` - the session handle of an active session.

Returns: nil

```
void cancelWorkItem(String workItemID, String sessionHandle) throws IOException
```

Purpose: Cancels the specified workitem. Sets its status as "Failed" - no further processing will occur on the same branch as the failed workitem.

Arguments: `workItemID` - the identifier of the workitem to cancel;
`sessionHandle` - the session handle of an active session.

Returns: nil

As mentioned previously, the Worklet Exception Service makes extensive use of *Interface X* and so is a good example to browse for tips and pointers.

That concludes this guide to creating a basic YAWL Custom Service. However, there are many more methods available via the Engine's Interfaces to support your Custom Service's operations. More information can be found in the "YAWL Book" [10], the YAWL javadoc (online at http://www.yawlfoundation.org/javadoc/yawl/), and by examining the code of the existing Custom Services distributed with each YAWL release.

Chapter 4

Implementing In-Process Custom Services

4.1 Introduction

This chapter describes an architecture, as an alternative to the REST-style, for connecting external applications *directly* to an instance of the YAWL Engine.

4.1.1 Why Use YAWL?

The justification for using the YAWL system in order to provide a "process flow controller" is simply that of extreme flexibility. There are numerous examples of coordinated software systems where some form of control flow exists, advancing the overall processing requirement from one stage to the next. A good example of such a control flow is that of a web browser session into a web server over the Internet. A user connects to some website in order to order some goods, and subsequently have them delivered. In this scenario, the user typically searches for and locates a number of items from the online stores catalogue, and then is directed to the payment details web form, and finally to the delivery details web form. Whilst there are several web development frameworks that support the simple routing of web pages, it is considered that such frameworks do not provide the rich set of process routing and data manipulation facilities found within the YAWL system.

In Enterprise Application Integration (EAI) scenarios, a separate set of features make the use of YAWL extremely attractive, especially when these are coupled with other Java2 Enterprise Edition (J2EE) features such as the Java Message Service (JMS). Traditionally, back-office or batch style applications are executed in some predefined sequence, controlled either manually or via complex shell scripts. In the situation where a Service Oriented Architecture (SOA) model is adopted[1], it becomes feasible that the sophisticated process control flow features of the YAWL runtime can be utilised to build highly automated and efficient back-office processing systems. In fact the YAWL runtime provides an extremely flexible and powerful routing engine. In an XML-centric integration architecture such as SOAP, the use of YAWL becomes even more beneficial given its comprehensive support for XML with its data manipulation and interrogation layer.

This chapter describes an architectural model utilising a single instance of the YAWL Engine to control multiple YAWL Custom Services. Each custom service effectively services a request into a back-end office application. The technique outlined in this chapter presents an architecture where each application may execute as a free-standing Java application with no requirement for an external application server[2]. The approach to connecting some custom services with their own external systems is left as an exercise for the

[1] Whereby a formalised interface is developed to allow external yet controlled access to application system functionality.

[2] This is unlike the default architecture for YAWL wherein a J2EE/Servlet container is used to coordinate a set of integrated Web applications.

reader, although various suggestions are made.

4.2 The InterfaceB Observer Gateway

4.2.1 Background

The YAWL Engine provides two primary interfaces that, one way or another, are exposed to the business environment:

Interface A	Management interface for loading and unloading process models, registering client applications and services, and observing case progress.
Interface B	An execution interface for launching cases (process instances) and worklist style operations such as starting and completing tasks within a case.

These engine interfaces listen and respond to requests from the business environment. Conversely the "Observer Gateway" interface is designed to allow external systems/modules to listen and respond to the YAWL runtime engine. Once an observer gateway instance is deployed into the YAWL runtime environment, the implementation may receive the notifications normally associated with YAWL custom services. As YAWL custom services typically implement the logic to process work item requests associated with atomic tasks, the events handled by the observer gateway are principally concerned with the scheduling and execution of tasks - like Interface B.

Thus, there are two alternative interaction models in communication with the YAWL runtime engine, as alluded to above. First there is a "pull style" model. In this model, the custom service polls the YAWL runtime on demand in order to retrieve, and ultimately display to the end user a list of outstanding work item tasks. The alternative is a "push style" model wherein the YAWL runtime engine can notify external systems of newly scheduled tasks etc. Since the Beta 7 YAWL version this "push style" model has been extended into a plug-in style architecture. The "Observer Gateway" interface forms a primary foundation to this framework. Now there is a conveniently flexible alternative to the XML/HTTP technique used to connect the YAWL runtime engine with custom services.

4.3 Interface Design

The "Observer Gateway" interface is designed to allow multiple Java objects to register interest with an instance of a YAWL Engine in order to receive notification as to when an atomic task's work item becomes available or unavailable. The object making the registration needs to provide a number of "listener style" methods as demanded by the observer gateway interface. As is usual for listener interfaces, the call into the listener method executes on YAWL's runtime thread and should not block execution for an extended period of time. It is suggested that the method implementing the listener interface creates and starts a separate worker thread to handle the request, thus allowing the runtime to continue with minimal delay. Whilst the Observer Gateway interface is used to connect with the custom YAWL services that are shipped with the system, it is open to integration with user developed code. Note that the `InterfaceB_EngineBasedClient` class, which generates event notifications to custom services, also implements the Observer Gateway interface.

In the existing HTTP based custom service model, (see Chapter 3) a Servlet container is used to host one or more custom services. Each custom service needs to be registered with the runtime via the YAWL Admin-

istration facility. Once registered, a custom service will be notified of new and cancelled work items via a well defined interface call. The observer gateway interface extends this model in the following ways:

1. The requirement to pre-register a service URL with the runtime is removed[3].
2. Assuming a suitable manager style application is used to interact with the Engine, custom services can be implemented where communication between it and the Engine (via a suitable manager) can be via any form of wire protocol.

4.4 Custom Service URLs

The existing YAWL custom service implementation utilises the configuration of specific URLs against an atomic task in order to route work requests (and hence notifications) to a specific custom service.

The same use of URLs is made within the observer gateway, but instead of routing on a one-to-one basis from an atomic task to a specific custom service, the "scheme" component of the URL is used to identify the object listening on the observer interface. Once consumed by the listener, the remainder of the URL specification can be used to forward processing of the work item's data associated with the atomic task using any scheme required by the user.

The observer gateway controller (implemented via class `ObserverGatewayController`) within the runtime Engine supports the registration of multiple objects which implement the ObserverGateway interface. The gateway then forwards requests to these registered objects using in-process method calls. It is therefore a requirement that both the Engine and all objects registering as observer gateway listeners **must be within the JVM instance**. Rather than expose the observer gateway to alternate protocols such as RMI, XML-RPC, SOAP, JMS etc., it is left as an implementation issue for the developer of the object which registers with the Engine. It is assumed that developers will design and implement a "Custom Service Manager" style application which is a sophisticated type of "Observer Gateway". It could perform the following functions:

- Register itself with the runtime specifying some unique "scheme" tag.
- Manage a pool of objects which implement the custom service processing.
- Invoke the required custom service calls using whatever protocol is required.

The advantage of the above approach is that developers are free to utilise whatever protocols they require, be they "standard" protocols found within the J2SE and J2EE stable, or custom protocols such as those required to communicate with legacy systems, possibly utilising non-Java based resources via the Java Native Interface language feature.

The URL routing between the standard custom service mechanism and that employed by the observer gateway is summarised in Figures 4.1 and 4.2.

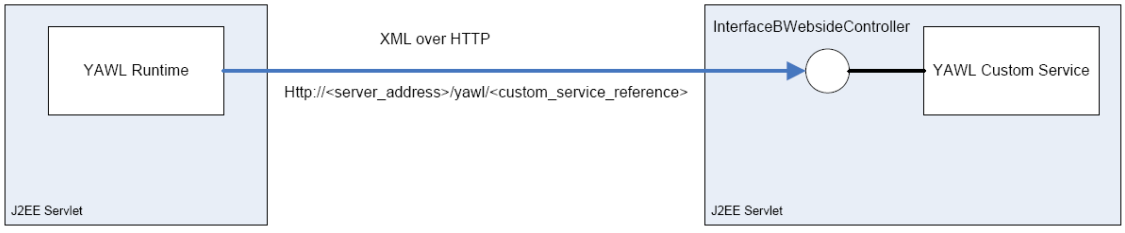


Figure 4.1: Classic style YAWL Custom Service Implementation

[3] Although pre-registration of such services within the runtime is not required, a warning message will be generated at both design time from within the YAWL Editor, and also at runtime when the containing specification is loaded.

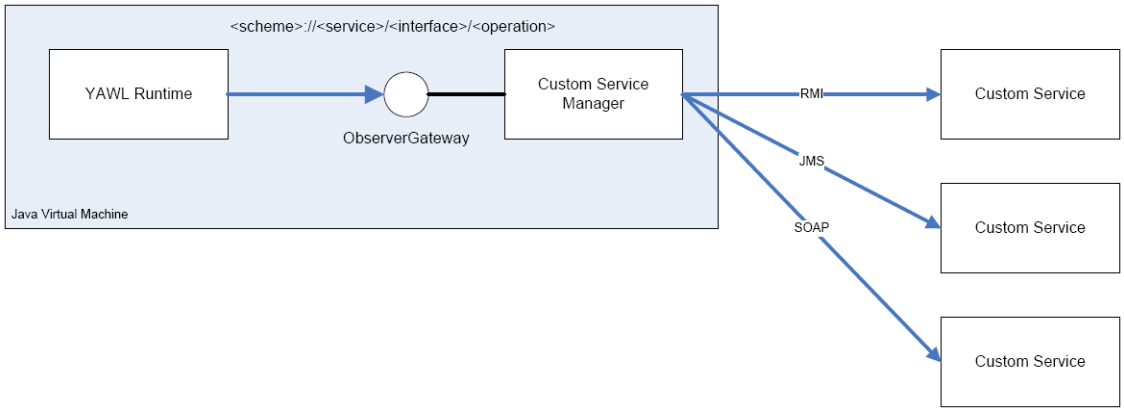


Figure 4.2: Observer Gateway YAWL Custom Service Implementation

In the classic style implementation, there is a one-to-one relationship between an atomic task (and hence work item) in the YAWL Engine and a specific custom service implementation. In addition, the custom service must be invoked using XML over HTTP protocol.

In the observer gateway architecture, the relationship between work items and custom services is one-to-many. This is achieved by employing a custom developer manager class which utilises some URL parsing scheme to determine which custom service to delegate the request to. How the URL is constructed in terms of path content and number of elements is completely open ended, the only constraint being that of adherence to standard URL formatting in terms of characters utilised. Such a solution generally adheres to the REST design principle, wherein each URI represent a particular resource.

A suggested URL formatting scheme is as follows:

```
<scheme>://service/interface/operation
```

The components are described in the following table:

Component	Description
scheme	This element defines the tag that links this URL to a specific scheme of custom service manager. This is the value that the manager uses when registering itself with the runtime for receipt of notifications on the observer gateway listener interface.
service	Describes the service as offered by the target custom service. “Order Processing” is a suitable example.
interface	Describes an interface (i.e. a collection of methods or operations) as supported by the “service”. “Provisioning” is a suitable example.
operation	Describes the operation of method to be called within the “interface”. “NewCreditOrder” is a suitable example.

4.5 Implementing the Manager Class

The use of a manager class is optional. Without an intermediate manager class, the object registering with the observer gateway will need to directly implement the business logic required to consume and process all work items where the URL associated with the workitem has the same URL “scheme” as that specified by the registering observer gateway.

For example, the following work item requests will be forwarded to observer gateways which register with a scheme of "xyz":

- xyz://sales/order/newOrder
- xyz://sales/order/cancelOrder
- xyz://inventory/stockitem/addNewStockItem

By making the registering Observer Gateway object a form of "manager", work item requests can be analysed and the work delegated to some other object or even external system. The routing can be performed by parsing the full URL path defined against the work item, or even by inspection of the XML data associated with the work item (this being available once the work item has been checked-out of the runtime - and hence moved from the "enabled" state to the "executing" state).

If we take the earlier URL examples, the use of a manager allows us for example to route all work requests for the "sales" domain to an external non-Java based system via some JNI wrapper interface. Work requests for the "inventory" domain however could be routed via a JMS interface to some external Java based application.

Internally within the Engine, all calls to register an object implementing the ObserverGateway interface result in a callback reference being stored within a table, and therefore many Observer Gateway objects can register with the Engine. When the Engine enables a new workitem, the atomic task definition associated with that workitem is examined and any associated custom service URL is extracted. The ObserverGatewayController within the Engine then examines all registered Observer Gateways, and if the URL scheme matches, a callback is invoked to inform all gateways for the given scheme (e.g. http) announcing the new workitem.

4.5.1 The ObserverGateway Interface

The custom service manager, or custom service itself (if no manager class is being utilised), must implement the `org.yawlfoundation.yawl.engine.ObserverGateway` interface. The interface is shown for YAWL 2.2 release[4] is shown in Listing 4.1.

getScheme

This method should return the scheme tag that this custom service manager supports (remember that a number of managers may register with the engine, so that each manages callbacks for one scheme tag). Suggested values are as follows, but are not validated in any way by the YAWL Engine, so that any scheme tag may be used:

jms	Java Method Call
rmi	Remote Method Invocation
http	Hyper Text Transfer Protocol
jms	Java Message Service
soap	Simple Object Access Protocol

announceFiredWorkItem

This method will be called by the YAWL Engine for each new workitem creation, where the scheme component of the service URL associated with that work item at design time matches that returned by a `getScheme()` call on an ObserverGateway. The invocation supplies a reference to a `YAnnouncement` object, a container for the YAWL service URL, the workitem, the event and its context (each has its own accessor method).

[4] **Note:** YAWL version 2.2 introduced some changes to the ObserverGateway interface.

```
1  public interface ObserverGateway {
2
3      String getScheme();
4
5      void announceFiredWorkItem(YAnnouncement announcement);
6
7      void announceCancelledWorkItem(YAnnouncement announcement);
8
9      void announceTimerExpiry(YAnnouncement announcement);
10
11     void announceCaseCompletion(YAWLServiceReference yawlService,
12             YIdentifier caseID, Document casedata);
13
14     void announceCaseCompletion(Set<YAWLServiceReference> services,
15             YIdentifier caseID, Document casedata);
16
17     void announceCaseSuspended(Set<YAWLServiceReference> services,
18             YIdentifier caseID);
19
20     void announceCaseSuspending(Set<YAWLServiceReference> services,
21             YIdentifier caseID);
22
23     void announceCaseResumption(Set<YAWLServiceReference> services,
24             YIdentifier caseID);
25
26     void announceWorkItemStatusChange(Set<YAWLServiceReference> services,
27             YWorkItem workItem,
28             YWorkItemStatus oldStatus, YWorkItemStatus newStatus);
29
30     void announceEngineInitialised(Set<YAWLServiceReference> services,
31             int maxWaitSeconds);
32
33     void announceCaseCancellation(Set<YAWLServiceReference> services,
34            YIdentifier id) ;
35
36     void shutdown();
```

Listing 4.1: The Observer Gateway Interface

announceCancelledWorkItem

This method will be called by the YAWL Engine whenever a previously announced work item for this service is cancelled. Again the call is only made where the scheme component of the task URL matches that returned by a `getScheme()` call on an ObserverGateway.

announceTimerExpiry

This method will be called by the YAWL Engine whenever a work item timer expires.

announceCaseCompletion - with Service Reference

Upon launching a case the caller may elect any Custom YAWL Service to be notified once the case is complete. When doing so, this method gets called by the engine once the case has completed. As usual, the call is only made for ObserverGateway instances where the scheme component of the URL matches that returned by a `getScheme()` call. The call supplies a reference to the YAWL service from which the URL can be obtained, together with a case ID, and the output case data.

announceCaseCompletion - without Service Reference

This version of the case completion method is called for cases where a completion listening service is not specified.

announceCaseCancellation

This method is called by the Engine to announce when a case is cancelled.

announceCaseSuspending

This method is called by the Engine to announce when a case starts to suspend (i.e. enters the 'suspending' state as opposed to entering the fully 'suspended' state).

announceCaseSuspended

This method is called by the Engine to announce when a case suspends (i.e. becomes fully suspended as opposed to entering the 'suspending' state).

announceCaseResumption

This method is called by the Engine to announce when a case resumes from a previous 'suspending' or 'suspended' state.

announceWorkItemStatusChange

Notifies a change of status for a work item.

announceEngineInitialised

This method is called by the Engine post-startup to advise that it has completed its initialisation routines and is in a 'running' state.

```
1    YEngine engine = YEngine.getInstance(false);
2    engine.registerInterfaceBObserverGateway(this);
```

Listing 4.2: Dynamically registering an Observer Gateway instance

shutdown

This method is called when the Engine is shutdown (i.e. when its servlet is destroyed); the observer gateway should use this notification to do its own finalisation processing.

4.6 Registering with the YAWL Engine

Any custom service or custom service manager instance needs to register itself with the Engine once it has completed its initialisation and is ready to accept requests. There are two way to register an observer gateway instance with the Engine:

1. In the Engine's `web.xml` file, enabling the `ObserverGateway` context parameter (it is commented out by default) and providing the fully qualified class name of the observer gateway instance as the parameter's value. Note that several observer gateways can be specified here, with each fully qualified class name separated by semi colons (;).
2. The object which implements the ObserverGateway interface needs to obtain a reference to the YAWL Engine and invoke the registration method as shown in Listing 4.2.

Once registered, the object will receive notifications for work items as they are generated. It should be noted that work items already posted against the customer service will not be re-notified, it being the responsibility of the manager to retrieve any outstanding work items and re-process these if required. To avoid such a situation, a reliable messaging framework such as JMS could be utilised to ensure requests in the form of messages are delivered to the target custom services, and avoid any requirement to re-post messages after a restart of the YAWL runtime and/or custom service manager instances.

Chapter 5

The Worklet Service

The Worklet Service provides an API that allows external custom services and applications to interact with it, and an event server that allows *listener* services to be notified when an worklet is selected or an exception raised.

5.1 Worklet Gateway Client

The `WorkletGatewayClient` is a client side class that can be used by custom services to interact with the WorkletService via an API. It provides two main functionalities:

- creating, accessing, updating and evaluating Ripple Down Rules (RDR); and
- adding and removing worklet event listeners (see Section 5.2).

An instance of `WorkletGatewayClient` can be used by custom services and applications to first create a session with the Worklet Service, then use the returned *session handle* in subsequent API method calls. Any service or application registered with the engine is trusted by the Worklet Service, and so the same credentials can used to establish the connection (see Listing 5.7 for an example of how to connect and use the API). Like the Engine interfaces, the client provides `connect`, `checkConnection` and `disconnect` methods (see also Section 3.2.2).

The client provides a number of methods for working with RDR maintained within the Worklet Service. Several helper classes are also used by the client, and so your custom service will need access to them. While the individual classes may be included in your service's war file, an easier and more convenient alternative is to place the *yawl-lib.jar* file, which contains *every* YAWL class, on your service's classpath (cf. Section 3.3.2).

5.1.1 Working with Ripple Down Rules

To begin, please see the Worklet Service chapter of the YAWL User Manual for background information on RDR and their use within the Worklet Service.

Since the primary aim of the Worklet Service is to support processes enacted by the YAWL engine, each rule set is uniquely identified by a `YSpecificationID` (that is each rule set is defined for a particular specification). However, to generalise RDR support for other custom services and applications, rule sets can also be uniquely identified by a process name (in fact, any name string can be used). Thus, for each client method there are several variations, to support the fact that either a specification id or a generalised process name can be used to identify rules. Please see the `WorkletGatewayClient` javadoc for more details.

```
1  public void example throws IOException {
2      YSpecificationID specID = new YSpecificationID(null, "0.1", "someSpecification");
3      String processName = "someProcess";
4      RdrMarshal marshal = new RdrMarshal();
5      WorkletGatewayClient client = new WorkletGatewayClient();
6      String handle = client.connect("admin", "YAWL");
7
8      // get a Rule Set
9      String ruleSetXML = client.getRdrSet(specID, handle);
10     if (! successful(ruleSetXML)) throw new IOException(ruleSetXML);
11     RdrSet ruleSet = marshal.unmarshalSet(ruleSetXML);
12
13     // get a Rule Tree
14     String ruleTreeXML = client.getRdrTree(processName, null,
15         RuleType.CasePreconstraint, handle);
16     if (! successful(ruleTreeXML)) throw new IOException(ruleTreeXML);
17     RdrTree ruleTree1 = marshal.unmarshalTree(ruleTreeXML);
18     ruleTreeXML = client.getRdrTree(specID, "Treat", RuleType.ItemSelection, handle);
19     if (! successful(ruleTreeXML)) throw new IOException(ruleTreeXML);
20     RdrTree ruleTree2 = marshal.unmarshalTree(ruleTreeXML);
21
22     // get a Rule Node
23     String ruleNodeXML = client.getNode(specID, null, RuleType.CasePreconstraint,
24         1, handle);
25     if (! successful(ruleNodeXML)) throw new IOException(ruleNodeXML);
26     RdrNode node1 = marshal.unmarshalNode(ruleNodeXML);
27     ruleNodeXML = client.getNode(processName, "Archive", RuleType.ItemSelection,
28         4, handle);
29     if (! successful(ruleNodeXML)) throw new IOException(ruleNodeXML);
30     RdrNode node2 = marshal.unmarshalNode(ruleNodeXML);
31 }
```

Listing 5.1: Examples of using the Worklet Gateway Client 'get' methods

Get Methods

Firstly, there are a number of *get* methods, which can be used to retrieve an individual rule node, a rule tree (a set of connected nodes for a particular rule type and task id), or an entire rule set (the set of rule trees for a particular specification or process). Each of the *get* methods, and in fact all of the client methods, return a String value representing a successful result or an appropriate error message. A successful result for a *get* method contains an XML String representation of the object retrieved, which can be unmarshalled into the appropriate object using the `RdrMarshal` class, as shown in the examples in Listing 5.1:

- A new `WorkletGatewayClient` is instantiated on line 5. Since no URL is passed to the constructor, the default URL for the worklet gateway is used (which assumes the Worklet Service is installed locally).
- Line 6 establishes a session with the client, and assumes the call will be successful, for simplicity. While the default 'admin' credentials have been used in the example, it is of course recommended that you use the actual credentials provided by your service when it was registered with the engine (cf. Section 3.3.4).
- Lines 9–11 retrieve a rule set for a specification from the Worklet Service, checks that the result was successful, then unmarshals the XML to a RdrSet object. A second version of the `getRdrSet` method accepts a process name instead of a specification id.
- Two rule trees are retrieved on lines 14–20. The first uses a processName and gets the *CasePreconstraint*

```
1  public void example throws IOException {
2      YSpecificationID specID = new YSpecificationID(null, "0.1", "someSpecification");
3      RdrMarshal marshal = new RdrMarshal();
4      WorkletGatewayClient client = new WorkletGatewayClient();
5      String handle = client.connect("admin", "YAWL");
6
7      // evaluate a Rule Type against a data set
8      String dataStr = "<data><Age>40</Age><Fracture>true</Fracture></data>";
9      Element data = JDOMUtil.stringToElement(dataStr);
10     String conclusionXML = client.evaluate(specID, "Treat", data, RuleType.
           ItemSelection, handle);
11     if (! successful(conclusionXML)) throw new IOException(conclusionXML);
12     RdrConclusion conclusion = marshal.unmarshalConclusion(conclusionXML);
13 }
```

Listing 5.2: Example using the Worklet Gateway Client 'evaluate' method

rule tree for it. Since we are retrieving a case-level rule tree, the second parameter, for task id, is passed a null value. The call on line 18 uses a specification id and gets the *ItemSelection* rule tree for its *Treat* task. Since we are retrieving an item-level rule tree, and each task for an item-level rule type has its own rule tree, the required task id value is supplied. There is also a variation of this method that takes a WorkItemRecord parameter, rather than a spec id and task id combination.

- Two rule nodes are retrieved on lines 23–30. Again, the call on line 23 is for a case-level rule type, so no task id is supplied, while one is supplied for the item-level rule call on line 27. We also pass the actual node id (integer) that identifies the node to be returned in each call. Again, there is also a variation of this method that takes a WorkItemRecord parameter, rather than a spec id and task id combination.

Evaluation Methods

The rule sets maintained by the Worklet Service can be evaluated against a given data set via the client API. Like the *get* methods, there are variations that accept a specification id or a process name, and if evaluating an item-level rule tree, a task id. There are three possible results when a rule set is evaluated:

1. No rule tree exists for the given specification id or process name, and/or task id, and rule type (as required). An error message to that effect is returned.

2. A rule tree exists for the given parameters, but none of its nodes' conditions were satisfied. An error message to that effect is returned.

3. A rule tree exists for the given parameters, and at least one its nodes' conditions were satisfied. An `RdrConclusion` is returned.

The conclusion of a rule node represents a set of *primitives*, each comprising an *action* and a *target*. For example, the conclusion of an *ItemSelection* rule node contains one primitive, with an action of 'select' and a target naming the worklet to select. The conclusion of an exception rule type may contain any number of primitives (see the Worklet Service chapter of the YAWL User Manual for more information). An `RdrConclusion` object is a convenience object that stores the primitives and allows you to iterate through them. Of course, if you are using the Worklet Service merely as an RDR store, and don't intend to have the rules evaluated by the Worklet Service in relation to a YAWL process instance, then the conclusion of a rule node can contain any data of relevance to your service or application.

An example of an *evaluate* method call is shown in Listing 5.2.

```
1  public void example throws IOException {
2      YSpecificationID specID = new YSpecificationID(null, "0.1", "someSpecification");
3      RdrMarshal marshal = new RdrMarshal();
4      WorkletGatewayClient client = new WorkletGatewayClient();
5      String handle = client.connect("admin", "YAWL");
6
7      // add a Rule Node
8      String cornerStr = "<data><Age>40</Age><Fracture>true</Fracture></data>";
9      Element eCornerstone = JDOMUtil.stringToElement(cornerStr);
10     RdrConclusion conclusion = new RdrConclusion();
11     conclusion.addPrimitive(ExletAction.Suspend, ExletTarget.Case);
12     conclusion.addCompensationPrimitive(''myWorklet'');
13     conclusion.addPrimitive(ExletAction.Continue, ExletTarget.Case);
14     RdrNode node = new RdrNode("Age > 75", conclusion, eCornerstone);
15     String newNodeXML = client.addNode(specID, "Treat", RuleType.ItemPreConstraint,
           node, handle);
16     if (! successful(newNodeXML)) throw new IOException(newNodeXML);
17     node = marshal.unmarshalNode(newNodeXML);
18 }
```

Listing 5.3: Example using the Worklet Gateway Client to add a new Rule Node

Adding a Rule Node

RDR are structured in such a way that rule nodes can be added at any time, but never deleted, since deleting a node would 'break' the tree's internal framework (again, more details on this concept can be found in the Worklet Service chapter of the YAWL User Manual). The client API provides a method to add a rule node to a rule set for a specification or process, with similar variations to those mentioned above for the different rule types. An example of adding a node is shown in Listing 5.3.

When adding a rule node, please keep in mind the following:

- The node being added requires values for *condition*, *conclusion* and *cornerstone* data only, as shown in lines 8–13 of Listing 5.3. Values for the node's *id*, *parent*, *trueChild* and *falseChild* fields are not required, and in any case will be replaced by the Worklet Service in the process of inserting the node in the appropriate location in its rule tree. A value for the *description* field is optional.

- The *condition* value is the condition that will be evaluated for the node when it is reached during a tree evaluation. It is important when framing conditions to be aware of how an evaluation traverses the tree, that is when a condition is satisfied for a node, that node's child node on its true branch (if any) is then evaluated; if it is not satisfied, then its false child (if any) is evaluated. Therefore, each node's condition forms a 'unit' in a larger, conjunctive condition, and so each node's condition should not repeat those parts that have already been evaluated in higher level nodes of the tree. For example, if a node's condition "Age $>$ 25" is satisfied, then the node being added as its true child might be "Height $<$ 170", rather than "Age $>$ 25 and Height $<$ 170", since the first part of the condition has already been evaluated and satisfied.

- The *cornerstone* data is very important when adding a node to an existing tree, as it is used to determine the proper location of the node. It must be valid XML, but the name of its root element can be anything – the name 'data' is a convention rather than a rule.

- The *conclusion* creation shown on lines 10–13 follows the required structure for conclusions intended to be executed by the Worklet Service (that is, a series of primitives, each with child 'action' and 'target' elements), but as stated above if you are using the RDR for other purposes then this format doesn't need to be followed, but should be a JDOM Element named "conclusion" with content appropriate it the purpose in use.

- The node returned by the call to the `addNode` method is the node added, complete with the added value for *id*.
- If no rule set currently exists for the specification id or process name given, then one is created.
- If no rule tree currently exists for the specification id or process name, and rule type (and task id if required) given, then one is created, and then the node is added as the first node in the tree (after the root node).

5.1.2 User-defined Functions

The condition defined for each rule node is an expression that must evaluate to a final boolean (true/false) value. Expressions can be formed using the usual numeric, comparison and logical operators (&, !, |), and operands consisting of literal values, and/or the values referenced by the name of case and/or workitem data variables. In more complex expressions, the usual order of precedence applies, and sub-expressions can be grouped with parentheses to override the order.

Alternately, a condition can be defined as an XQuery expression, which is useful if you need to query case or workitem data stored as complex data types. An XQuery should be enclosed in curly braces, for example: {`/myTask/Person/Name/text() = 'John'`}.

In addition, conditional expressions support the use of *user-defined functions*, that is functions that are defined to evaluate the available data in a particular way, which can then be inserted into expressions. These functions can pass arguments consisting of literal values, and case or workitem data variable names, which will be substituted with their actual runtime values before being passed to the function.

There are two ways to add user-defined functions to the condition evaluator. The first is detailed in Section 8.7 of the YAWL User Manual, which involves adding code to an existing class and recompiling it. The second, perhaps neater way, is described below.

Adding a Pluggable User-defined Function

This method involves creating and adding pluggable classes to the service's classpath.

Step 1. Defining the Function Each function is defined in its own class, which implements the java interface `RdrFunction` . There are two methods to implement (Listing 5.4):

- `String getName()`, that should return the name of your function (which can be different to the name of the new class). The name returned will match the name used within a conditional expression for a rule node.
- `String execute(Map<String, String> args)` is the method where the body of your function is defined. The Map of arguments passed in are String key-value pairs, each pair consisting of the name of the variable stated in the function and its (current) value. Any non-String values are converted to Strings as required. This method must return a String value, which may be any value if the expression uses the function as part of a comparison operation, or "true" or "false" if the expression uses the function as a standalone value or as part of a logical operation. For example, if the expression is `myFunc(x) = ''Confirmed''`, then your function should return a String value such as "Confirmed" or "Denied"; if the expression is `myFunc(x) | isActive(this)` or simply `myFunc(x)`, then your function should return "true" or "false". If your function is used as part of a numeric comparison, for example `myFunc(z) < 50`, your function should return its String equivalent, which will be interpreted as a numeric by the Condition Evaluator.

For work item level rules, a special argument, called `this`, can be passed via your function call (e.g. `myFunc(this)`). It's value will be an XML string of the complete work item record, so that specification and task descriptors, and state and timing values can be used by your function.

```
1  package com.example;
2
3  public class MyRdrFunction implements RdrFunction {
4
5      public String getName() { return "myFunc"; }
6
7      public String execute(Map<String, String> args) {
8          String x = args.get(''x'');
9          return x.equals(''OK'') ? ''Confirmed'' : ''Denied'';
10     }
```

Listing 5.4: An example RdrFunction implementation

Step 2. Creating a Jar Once your RdrFunction implementation is complete, it must be bundled up in a jar file, using the format shown in Figure 5.1. Under a working directory (of any name), place the compiled class that implements the RdrFunction interface in its package structure. For example, in Figure 5.1 our class is in the `com.example` package, so it is placed in a *\com\example\MyFuncImpl.class* directory path. Also under the working directory, create a subdirectory called `META-INF` to contain the manifest file for the jar, then under that create a subdirectory called `services`.

In the `services` file, create a new, plain text file and name it as the full name of the RdrFunction interface, namely `org.yawlfoundation.yawl.worklet.support.RdrFunction` (no txt extension). In that file, enter the full name of your implementing class on a single line (for example, `com.example.MyFuncImpl` – no class extension). That is, the contents of the text file should contain the fully qualified class name of your implementation. If you have implemented several RdrFunction classes, they should each be listed in this same file, one on each line.

Figure 5.1: RdrFunction implementation JAR structure

To compile the directory structure into a jar (of any name), enter this command from a prompt in the working directory: `jar cvf somename.jar -C ./ .`

Step 3. Jar to classpath The final task is to place the new jar file on the Worklet Service's classpath. The easiest way to do this is to copy it to the jar to the *tomcat\webapps\WEB − INF\lib* directory (create the *lib* directory if it doesn't already exist). Restart the service and your new function will be ready for inclusion in conditional expressions.

```
1  public class ExampleWorkletListener extends WorkletEventListener {
2
3      public void caseLevelExceptionEvent(String caseID, Element caseData,
4              RdrNode node, RuleType ruleType) {   }
5
6      public void itemLevelExceptionEvent(WorkItemRecord wir, Element caseData,
7              RdrNode node, RuleType ruleType) {  }
8
9      public void selectionEvent(WorkItemRecord wir, Map<String, String> caseMap,
10             RdrNode node) {  }
11
12     public void constraintSuccessEvent(String caseID, WorkItemRecord wir,
13             Element caseData, RuleType ruleType) { }
14
15     public void shutdown() { }
16
17 }
```

Listing 5.5: An example Worklet Event Listener implementation

5.2 Worklet Event Listeners

Each time the Worklet Service selects a worklet for a work item, or raises an exception for a work item or case, details of the event are announced to registered listener classes. Listeners can use these event announcements to perform additional processing as required, and make it possible to create *exception handling service chains*. Note that if the Worklet Service's exception handling facility is configured as disabled, no exception announcements will be generated (see the Worklet Service chapter of the YAWL User Manual for details on configuration).

5.2.1 Creating a Worklet Event Listener

An abstract class called `WorkletEventListener`, found in the `worklet.support` package, is the base class for all worklet listeners. This class is a `HttpServlet` that handles all of the mechanics involved in receiving notifications from the Worklet Service and transforming them into five abstract method calls that are to be implemented by extending classes. Listing 5.5 shows a basic example of a worklet event listener implementation.

The five methods are:

- **caseLevelExceptionEvent** is called each time a case level exception is raised. The `caseData` parameter contains the case-level data that was current when the exception was raised and was used to determine the rule violation. The `ruleType` parameter describes what type of exception has been raised; it will match one of the three case-level exception types (pre-constraint, post-constraint, external trigger). The `node` parameter is the RdrNode that evaluated to true and so triggered the raising of the exception.
- **itemLevelExceptionEvent** is called each time a item level exception is raised. The `ruleType` parameter describes what type of exception has been raised; it will match one of the three case-level exception types (pre-constraint, post-constraint, in situ constraint violation, external trigger, timeout, resource unavailable, work item abort). The `node` parameter is the RdrNode that evaluated to true and so triggered the raising of the exception.
- **selectionEvent** is called each time a worklet selection occurs, that is when a worklet is substituted for a work item. The WorkItemRecord will contain the usual case, task, work item and data descriptors. The `caseMap` parameter is a map of key-value pairs, each pair consisting of the case id and worklet

```
1   <servlet>
2     <servlet-name>WorkletEventListener</servlet-name>
3     <servlet-class>
4         org.yawlfoundation.yawl.myService.ExampleWorkletListener
5     </servlet-class>
6     <load-on-startup>1</load-on-startup>
7   </servlet>
8
9   <servlet-mapping>
10    <servlet-name>WorkletEventListener</servlet-name>
11    <url-pattern>/workletlistener</url-pattern>
12  </servlet-mapping>
```

Listing 5.6: Listening service's web.xml addition

name for each worklet selected for the work item — one case for single-instance tasks and several cases for multiple-instance tasks (a worklet is selected for each instance of a multiple-instance class, and a different worklet may be selected for each instance). The `node` parameter is the RdrNode that evaluated to true and so triggered the raising of the selection.

- **constraintSuccessEvent** is called each time a case or work item, that has pre- or post-constraint rules defined, has had these rules evaluated and the case or work item passes the constraints (i.e. no rules were satisfied and so exception was raised).
- **shutdown** is called when the Worklet Service is shutting down, and allows implementers to take any necessary action.

Once the listener class implementation is complete, an extra servlet definition section needs to be added to the web.xml of your service, as shown in Listing 5.6. The `<servlet-name>` values on lines 2 and 10 can be any name, but must match. On line 4, the fully-qualified name of your service's event listener implementation class should be entered. Line 11 contains the URL that will form the resource name that references the listener implementation, and again can be any name. For example, if your service is installed locally and is named as per the listing, then the full URL to it will be `http://localhost:8080/myService/workletListener`.

The final step is to register your listener with the Worklet Service. Listing 5.7 shows an example of the code required to do that (see also Section 5.1). Regarding the example:

- On line 4, a URL String of our listener is given. Note that it matches the URL defined via our web.xml settings above.
- Line 6 contains the call to add the listener to the Worklet Services set of listeners. The `addListener` method returns a String containing a success or error message, which is converted to a boolean value via the `successful` method (which will be available assuming that your custom service extends `InterfaceBWebsideController`). Added listeners can be removed from the Worklet Service via a `removeListener` method.

Your service should first ensure that the Worklet Service has completed its initialisation before the registration is attempted. The easiest way to do that is to have your custom service implement Interface B's `handleEngineInitialisationCompletedEvent` (see Section 3.4.2) – the Engine completes its initialisation moments after the Worklet Service does.

```
1     public boolean registerWorkletListener() {
2         WorkletGatewayClient client = new WorkletGatewayClient();
3         try {
4             String url = "http://localhost:8080/myService/workletlistener";
5             String handle = client.connect("admin", "YAWL");
6             return successful(client.addListener(url, handle));
7         }
8         catch (IOException ioe) {
9             log.error("Failed to  register listener: " + ioe.getMessage());
10        }
11        return false;
12    }
```

Listing 5.7: Listening service's web.xml addition

Chapter 6

Custom Forms

When a work item is listed with *Started* status on a participant's work list, the participant may choose to view the work item and, in doing so, review input values and supply output values for the input and output variables associated with the work item. By default, the Resource Service's *Dynamic Forms* component will use the work item's parameters to compile a data schema of the work item, using the type definitions of the parameters, and use the schema to dynamically generate a suitable web form that the user can interact with. This automated form generation removes the need for designers to consider how work will be displayed to users.

Dynamic forms are designed for maximum flexibility and are able to display input fields for work item data of any type. However, the generic look and feel of the generated form may be less than ideal in scenarios where a more professional presentation is required, or where organisations have standardised UI look and feel requirements. While the use of extended attributes goes a long way towards handing some control back to UI designers, there is a limit to what can be achieved.

In situations where full control over form presentation is required, the Resource Service allows designers to choose a *custom form* to associate with a task. At runtime, a work item that has a custom form associated with it will present that form to the user, in precedence to a dynamically generated one. While Chapter 4 of the YAWL User Manual describes how to associate a custom form with a task at design time, the purpose of this section is to describe how to implement a custom from so that it can (i) be populated with the relevant task input data; (ii) update the work item's output data with the form's contents when it is submitted; and (iii) return control to the calling work list.

A custom form can be written in any language that supports the ability to parse request parameters from a URL, provide user interaction, and make HTTP requests via an interface. The running example in this section uses a JSP form, but the description is not limited to JSP. In fact, the form's content (layout, input fields, buttons etc.) is irrelevant for the purposes of this section – the focus is on the mechanism to receive, update and return work item data. The example is contained in a file called `customformexample.jsp` which can be found in the root directory of an unpacked `resourceService.war` file (i.e. in the directory `tomcat/resourceService`), and contains detailed comments to describe the various steps.

NOTE: Since the YAWL UI component is responsible for the actual displaying of forms, it acts as a conduit for all interactions between custom forms and the Resource Service; that is, all custom form API endpoints are located in the UI component.

6.0.1 Step 1: Initialisation

At design time, a custom form is specified for a task by adding the URL of the form to a task property. At runtime, the Resource Service checks this field and, if populated, redirects the browser to the URL provided. Of course, the form has to actually exist and be accessible at the specified URL at the time the redirection occurs; if there is a problem accessing the form, the service will fall back to showing a dynamic form for the

```
1     // get the workitem xml. If the form has refreshed, it will be stored in a session
2     // attribute (see below). If not, get it from the gateway
3     String itemXML = (String) session.getAttribute("itemXML");
4     if (itemXML == null) {
5         String itemid = request.getParameter("workitem");
6         String participantid = request.getParameter("participantid");
7         String handle = request.getParameter("handle");
8         String callback = request.getParameter("callback");
9
10        session.setAttribute("workitem", itemid);
11        session.setAttribute("participantid", participantid);
12        session.setAttribute("handle", handle);
13        session.setAttribute("callback", callback);
14
15        // Use the callback with a GET method to retrieve an xml record of the workitem
              .
16        itemXML = send(callback, "GET");
17        session.setAttribute("itemXML", itemXML);
18    }
```

Listing 6.1: Retrieving the work item record

work item.

In addition to the URL specified at design time, the service adds the following parameters:

1. `workitem` – the identifier for the work item initialising the custom form.
2. `participantid` – the identifier of the currently logged on participant (i.e. the one viewing the form). Note that this is an internal identifier, not the participant's user id.
3. `handle` – the participant's current session handle.
4. `callback` – a pre-filled URL of the UI component's custom form API from which the custom form is being called, that includes the `handle` and `work item id` parameters.

Where static parameters have added to a custom form URL at design time, the four additional runtime parameters are appended after any static ones.

Append the endpoint method *item* to the callback URL, then send it with a GET verb to retrieve the WorkItem-Record for the work item (see 6.1).

6.0.2 Step 2: Parsing the work item data

Once you have accessed the work item data string, you can parse it for the relevant values to display on the form. Work item metadata is described in the various elements of the record, while the actual work item data parameters and values are stored in the `data element`.

To parse the work item record String into its equivalent XML form, any appropriate XML parser can be used. In our example, we convert the String into an JDOM element, which assumes a `jdom.jar` on the classpath (see Listing 6.2). The `data` element is a child node of the root element. Parse the data Element to display the relevant data values on the form.

Other GET endpoints :

- Append the *parameters* endpoint to the callback URL to retrieve the set of parameters (i.e. input- and output- task variables) for the work item.

```
1    Element wir = new SAXBuilder().build(new StringReader(itemXML)).getRootElement();
2    Element data = wir.getChild("data");
3
4    // one level down from data is the actual workitem data tree
5    Element wirData = (Element) data.getChildren().get(0);
```

Listing 6.2: Parsing the work item record

- Append the *outputOnlyParameters* endpoint to the callback URL to retrieve the set of output parameters (i.e. output task variables) for the work item.

6.0.3 Step 3: Saving / Completing the updated work item data

To save the user-updated data values, replace the relevant values in the data Element created in step 2, then send the String equivalent of that data Element back to the UI component, using the callback URL with the *save* endpoint appended, via a POST method. Once the data is saved, the work item can be (optionally) completed by POSTing the callback, appended with the *complete* endpoint.

Chapter 7

The Editor

7.1 Introduction

This chapter describes the technical aspects of the YAWL Editor (currently at version 2.1). Knowledge of Java, UML and the semantics of YAWL is assumed. The chapter is intended to give the reader an introduction and insight into to the editor's design. While the reader may find some features of the editor's design and implementation somewhat "odd", they are nevertheless that way for sound reasons. Many are legacy concepts from the initial desire to build a quick prototype.

7.2 Foundations

The editor has been written in such a way that the core graphing library, JGraph, forms the very heart of the application. A decision to replace the core graphing library is a decision to rewrite the editor from scratch. Its reach goes too far, and is too tightly coupled to the rest of the code for the editor to be considered anything other than an "application of JGraph".

Section 7.2.1 covers the motivation for, and basics of using JGraph. Section 7.2.2 gives a high-level tour of the Java packages of the editor, introducing the nature of classes that are to be found in each package and highlights packages of particular importance. Section 7.2.3 introduces how the base JGraph library has been extended to achieve the YAWL editor's key graphing functionality. Section 7.2.4 describes interesting aspects of the editor's build script.

7.2.1 JGraph

At the inception of the YAWLEditor project, a number of desired requirements for the editor were listed by the drivers of the project. The editor was to be:

- freely available,
- runnable heterogeneously,
- easy to develop,
- inter-operate easily with the YAWL Engine,
- easy to deliver and run.

The greatest impediment to delivering a prototype IDE for YAWL was that of an adequately powerful graphing library. The project could only show quick outcomes by relying on a third-party graphing library that also enabled these requirements.

At the time, the JGraph drawing package was the only package to meet these requirements from the perspective of a drawing environment for YAWL. A close second was the GEF drawing framework that comes with the Eclipse IDE. The final decider against GEF, however, was the requirement that the editor be a plugin to Eclipse, and that users of the editor would need Eclipse as a base environment to run within. The authors of GEF stated at the time that it might be possible to run GEF independently of Eclispe, but that those who were to try were on their own if they encountered problems. While building standalone Eclipse applications is now commonplace, at the time of the original design of the editor (2004), it was decided that JGraph was the better approach.

The requirements are re-listed below with a brief description per requirement of JGraph's match to the requirement:

Freely available: Written in Java, released under LGPL. There were no runtime environment licensing costs for the language, and the library was free to use in programming efforts so long as the LGPL requirements were respected.

Runnable heterogeneously: Java programs allows a high likelihood that they will run with little to no changes on different operating systems with compatible JREs. Also, few heterogeneous languages besides Java and C# had a rich enough GUI environment to be considered viable.

Easy to develop: Again, Java is one of the best supported programming languages in terms of learning and adoption. Though the library was not exactly easy to develop with, it was the only free stand-alone Java library being actively supported at the time.

Inter-operate easily with the YAWL Engine: The engine was written in Java, and to leverage as much of what the engine had to offer as possible, the path of least effort was to call engine API methods in the same language.

Easy to deliver and run: JGraph could be incorporated into an application to produce a standalone Java application. GEF, which shared many of the above requirements until this point required the Eclipse IDE to run, and expected an application using GEF to be written as an Eclipse plugin. At the time, this was not a trivial exercise to accomplish. Opinion by the project owners at the time was that they also disliked the idea of requiring such a large IDE as a pre-requisite for trying the editor.

JGraph can be a difficult library to understand initially. The reader is directed to the JGraph tutorial for an official discussion of the framework. Below is a brief, bottom-up description of the environment. A small class diagram of the "'core" JGraph model primitives[1] is supplied in figure 7.1.

A graphing package at its heart is a convenience package allowing a user to easily connect a set of nodes with lines. In JGraph, a node is made by implementing the abstract `GraphCell` interface, and a line an `Edge`. The only way to connect an edge to a node is via another type of cell called a `Port`. (Nodes, edges and ports are all subtypes of the `GraphCell` interface). A port must be a child of a node. Thus, to connect one node to another with an edge, you do so via connecting that edge to a child port for each of the nodes. Figure 7.2 illustrates this relationship with a concrete example.

In figure 7.2, two tasks are connected to each other with a flow. The tasks are `YAWLVertex` objects that implement the `GraphCell` interface. The flow is an extension of `Edge` interface. The edge is selected, making visible the ports connecting the edge to the boundaries of the relevant tasks.

JGraph is an implementation of the MVC (Model-view-controller) software design pattern. An implementation of this pattern requires a model object, a controller object, and a number of view objects. Very basic versions of each type of class are supplied via `DefaultGraphModel`, `JGraph` and `GraphLayoutCache` for the model, controller and views, respectively. Implementations of the interfaces listed in figure 7.1 must be placed within a `DefaultGraphModel`, to be considered part of that graph.

An important conceptual note must be made here on how to appropriately use JGraph:

[1] Even then, this is not a faithful representation of the true relationships of classes within JGraph, but does act as a good model of how to view how you'd typically consider these key classes to interact.

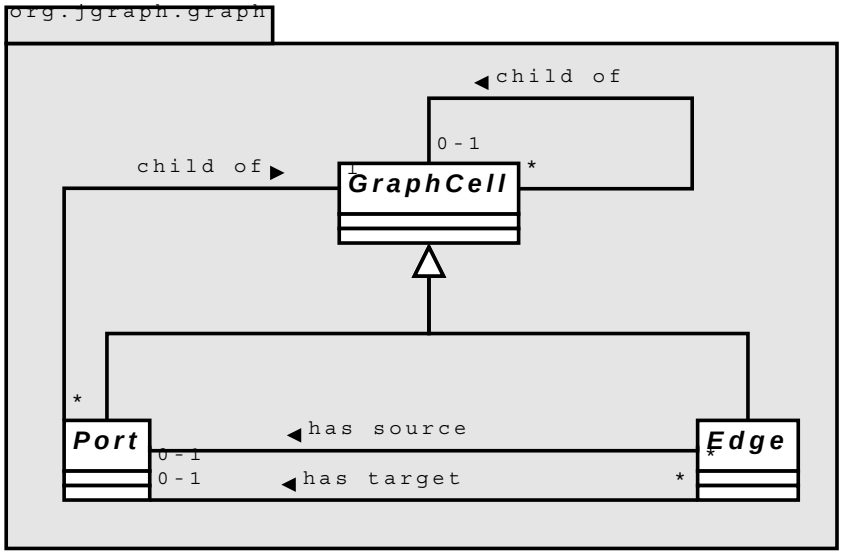


Figure 7.1: A class diagram of the primitive model classes of JGraph

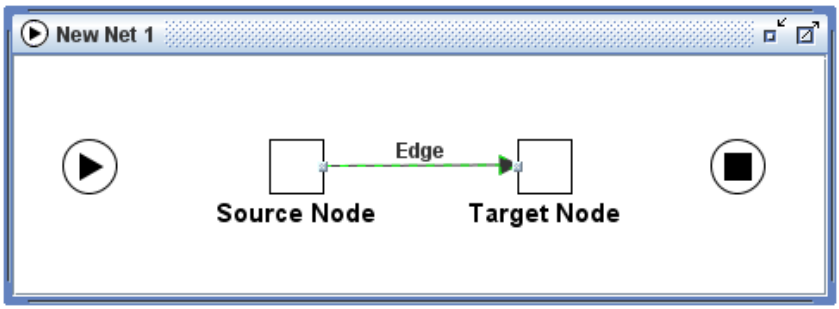


Figure 7.2: An example editor net, showing nodes, edges and ports.

A change to the graph model will only be recognised and responded to if done through key methods of the view, controller or graph model objects. If you directly change the color of a node, say, by programatically updating its colour directly, the controller will not know to inform the various views of that node that the colour has changed, and that they must now redraw themselves.

All changes made to a graph ultimately must go through one of the three methods of `DefaultGraphModel`: `insert()`, `remove()`, and `edit()`. As stated earlier, failure to make changes via these methods will leave the views out of sync with the base model. It will also not record those changes as undoable events, which is typically automatically handled by the JGraph framework. Note that these methods are also supplied on the view and controller classes. It is sometimes easier to use one of these classes, if that is the level that your own code has been working on. Note, however, that especially with the view class, things don't always work as expected. Some attribute changes will not be reflected in the model, if done from the view.

Changing attributes on JGraph objects warrants some explanation. When first building an object for placement in a JGraph, the programmer is expected to load a number of attributes into a `HashMap` container, and apply these attributes to the object via the map. The object can then be added to the graph via a call to the graph's `insert()` method. Example editor code is supplied in listing 7.1 to illustrate the point.

```
private void buildContent() {
  HashMap map = new HashMap();

  GraphConstants.setLineEnd(map, GraphConstants.ARROW_TECHNICAL);
  GraphConstants.setEndFill(map, true);
```

```
    GraphConstants.setLineStyle(map, GraphConstants.STYLE_ORTHOGONAL);
    GraphConstants.setBendable(map, false);
    GraphConstants.setEditable(map, true);

    GraphConstants.setDisconnectable(map, true);
    GraphConstants.setConnectable(map, true);

    getAttributes().applyMap(map);

    setPredicate("true()");
    setPriority(0);
  }
```

Listing 7.1: Setting attributes on a model object

Listing 7.1 shows code for setting key attributes of flow relations, which are a kind of edge within YAWL. The `GraphConstants` class is used to ensure that the hashmap is built only with type-safe data that is compatible with a given attribute.

Note also that a great deal of the API for JGraph requires that arrays of the most basic Java `Object` be passed around. The JGraph tutorial lists a number of reasons why they feel this is a necessary part of the framework. However, as a user of the framework, one should be careful not to come to grief by finding objects of an unexpected class mixed into a return set. Another side-effect of this is the large amount of class-casting code required on the boundaries between the editor code and the JGraph library.

Though there is a great deal more to the framework in terms of constructing views, listening for events, and the like, a discussion on such aspects is left until we actually describe key extensions made for YAWL. On a final note for JGraph, the reader is urged to familiarise themselves with the JGraph FAQ, and to become familiar with JGraph's forums for further support.

7.2.2 Package Contents

This section briefly describes the Java package hierarchy of the YAWLEditor, giving a basic guide to what a programmer may expect to find in any given editor source package. Any Java code that belongs specifically to the editor will reside in a sub-package of the base editor package `org.yawlfoundation.yawl.editor`. We assume that this package is reserved specifically for editor code. This base editor package deliberately contains only two classes:

`YAWLEditor.java`: The bootstrap class that starts the editor running.

`TestYAWLEditor.java`: This is the core JUnit test class that binds all other JUnit test classes for the editor so that they may be executed as a single unit test.

Figure 7.3 lists the current Java packages that makes up the core editor code, and briefly describes the nature of the classes to be found in each package. If there is a "key" package of the editor, containing classes that do the core job of graph manipulation, that package is `org.yawlfoundation.yawl.editor.net`. On a darker note, if there is anywhere in the editor where classes have tended toward implementing the god object anti-pattern, this this is the package for the very worst examples.

Another important package is `org.yawlfoundation.yawl.editor.specification`. This package, however, is focused on managing an entire specification. There is very little code specific to graphing in this package, but the classes here are responsible for saving and loading a specification, and managing specification-wide events.

In later sections, key classes from both packages will be described in much more detail.

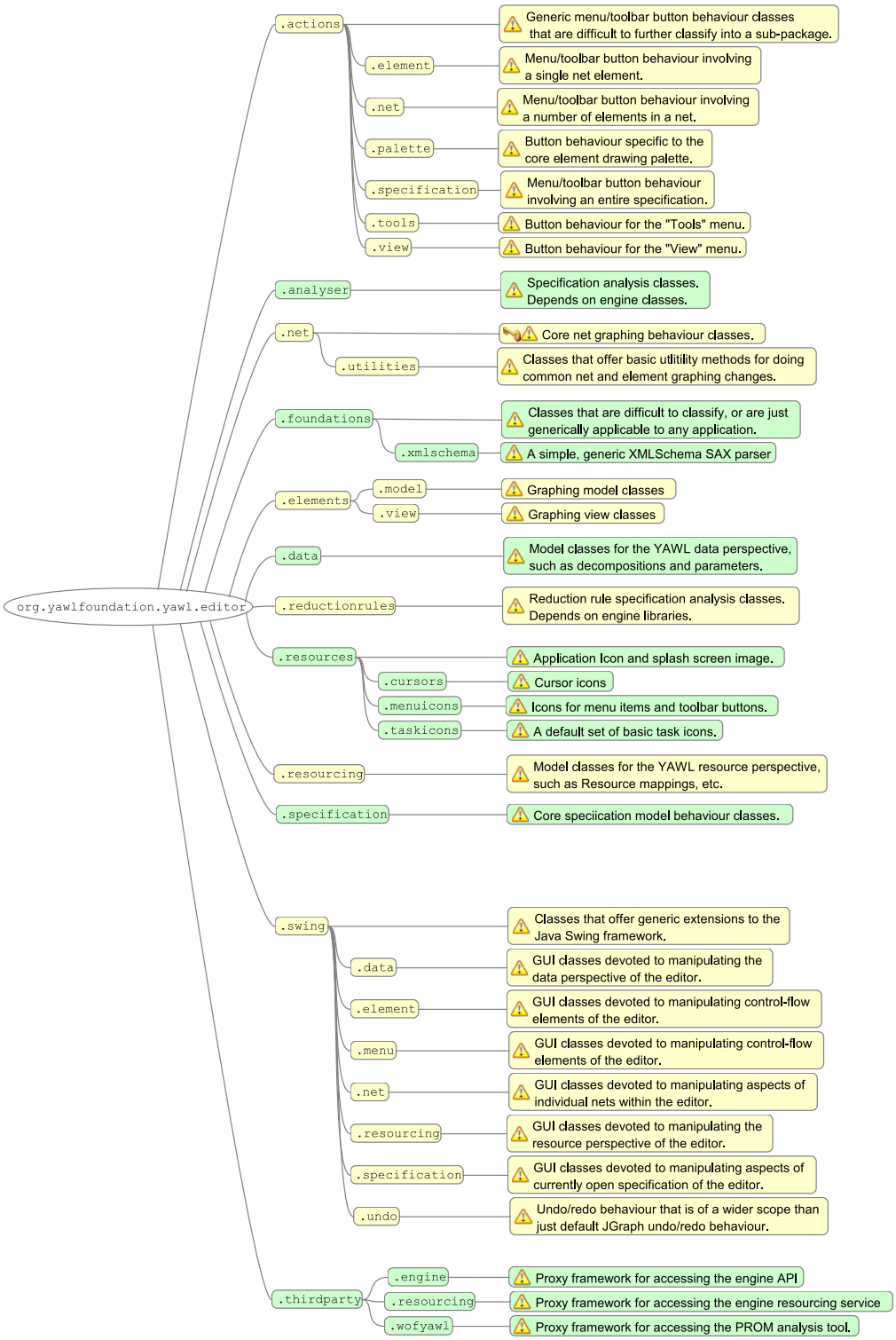


Figure 7.3: The Java package hierarchy of the editor

7.2.3 Extensions to JGraph

With a basic understanding of how JGraph works from section 7.2.1, we now describe how the basic framework of JGraph has been extended to implement the YAWLEditor. This section describes passive elements in section 7.2.3 and active elements in section 7.2.3 depending on whether they need further processing, or are responsible for the processing, respectively.

Passive Graphing Elements

Figure 7.4 is a class diagram showing how the graphing model constructs of JGraph (see figure 7.1) have been extended to implement the YAWLEditor graphing model. These constructs reside in the package `org.yawlfoundation.yawl.editor.elements.model`. The package is *passive* in the sense that it's not enough to just change attributes on objects in the package and have the change work. The changes must be processed through an *active* component, described later.

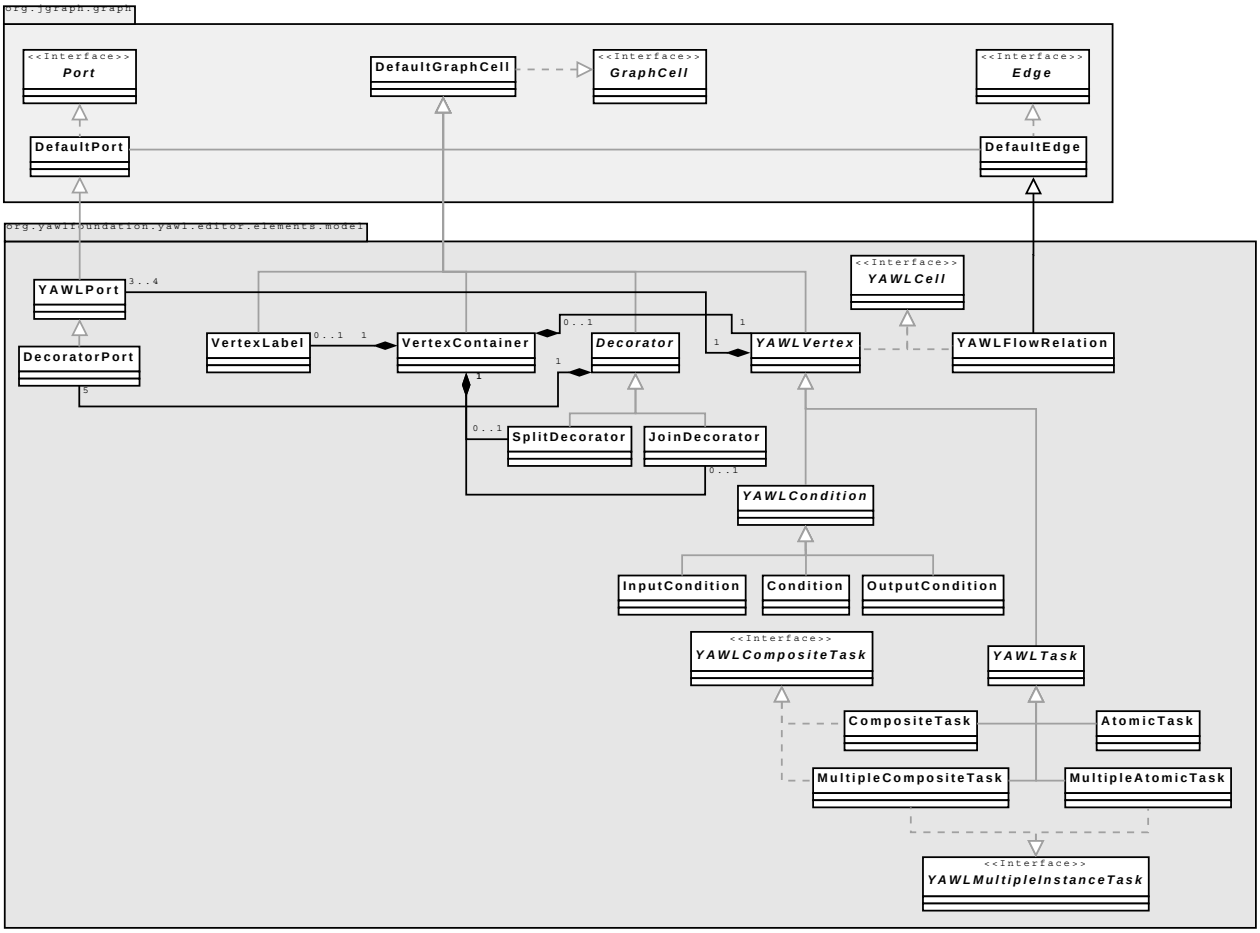


Figure 7.4: A class diagram of the graph model of the YAWLEditor

A few features of figure 7.4 need discussion. We can see from the class diagram that a number of classes inherit from `DefaultGraphCell`. The most important of these from a design perspective is `YAWLVertex`. The other three inherited classes (being `VertexLabel`, `VertexContainer` and `Decorator`) play a subservient role to `YAWLVertex`. For example, if a `VertexContainer` exists, it can only do so if it contains a `YAWLVertex`. Though other constructs can be contained within a `VertexContainer`, they are all optional, and very much depend on the existence of the vertex to have any meaning.

Note that the concept of a `VertexContainer`, and explicit `Decoratorator` classes for splits and joins

was a conscious design decision. An alternative was to have tasks contain all split and join behaviour. This alternative would have been a more natural fit to the engine native XML file format where there is no concept of a decorator for a task to describe a split or a join. After some consideration, it was felt that the complexity of a single `Task` object (especially view objects, described later) would be too high, and involve far too much coupling to a task's model and view objects, if it were also to contain split and join logic.

Figure 7.5 shows a concrete example of a `VertexContainer`, containing an `AtomicTask` instance, a `SplitDecorator`, a `JoinDecorator` and a `VertexLabel`. The container has been selected, and its boundary is currently visible for the purpose of manipulating the entire set of components within it as a single entity. The container's purpose is to hold together a number of graph constructs that need to be considered a single conceptual entity by a workflow designer. Again, it should be stressed that without the `AtomicTask`, there is no purpose to the other constructs that have been bound to it.

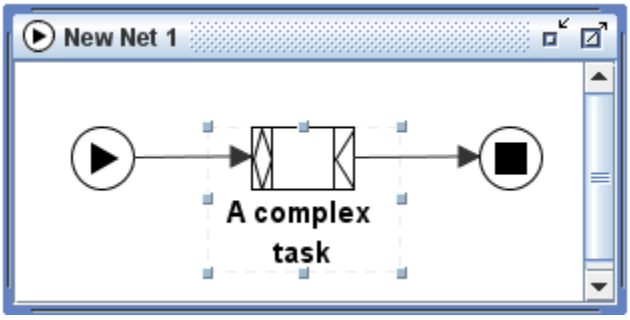


Figure 7.5: A concrete example of a VertexContainer binding task elements together.

Note also from figure 7.4, that there are a number of ports bound to certain key objects. A design decision was made to have a number of ports per net graph element, to be positioned along an appropriate boundary of the graph element. The alternative considered was to have a single port at the centre of a node where all flows for that node could be connected, which is more in keeping with packages such as Microsoft Visio. This Visio-style approach has also been occasionally asked for in the past. Two insights led the the approach constructed:

- Splits and joins in YAWL overlap in their representative icons. A single port would allow a great deal of ambiguity over the kind of split or join that a task had (a flow could be drawn running into an or-split, for example, which is also drawn the same way as an or-join). It was decided that to force only outgoing flow from splits, and incoming flows to joins, thus removing a degree of possible confusion around splits and joins.
- The mechanics for having a flow connect to the center of a node, but only drawn to its edge breaks down when a task has a split and a join on adjacent edges. The bounding-box of the entire task element set, including the split and join has a small square of whitespace between the two decorators, and flows would run only to the edge of this whitespace. The result would be a flow that seems to stop short of the task to which it's connected.

Conditions and tasks both have sub-trees of inheritance from `YAWLVertex` and each encode their own specific required model constraints. Note that the class hierarchy depicted in figure 7.4 introduces a set of methods that allows a particular class to answer questions about what may validly be done with it, meaning that as far as possible, rules are localised to the class involving them. For example, a standard `Condition` allows both incoming and outgoing flows. If you were to ask an `InputCondition`, however, if it allowed incoming flows, it would answer in the negative. The reader is directed to the interface `YAWLCell` for a set of basic methods that all tasks, conditions and flow relations must implement. Further interfaces (see `YAWLCompositeTask` for example) deeper in the hierarchy introduce more specialised methods to enforce requirements particular to certain workflow elements.

The corresponding views of the model objects described above can also be viewed as passive. They simply draw on the screen a representation of their corresponding model object equivalents. Figure 7.6 is a class

diagram mapping key model classes from figure 7.4 to the view classes responsible for their rendering. The view classes required reside in the package `org.yawlfoundation.yawl.editor.elements.view`.

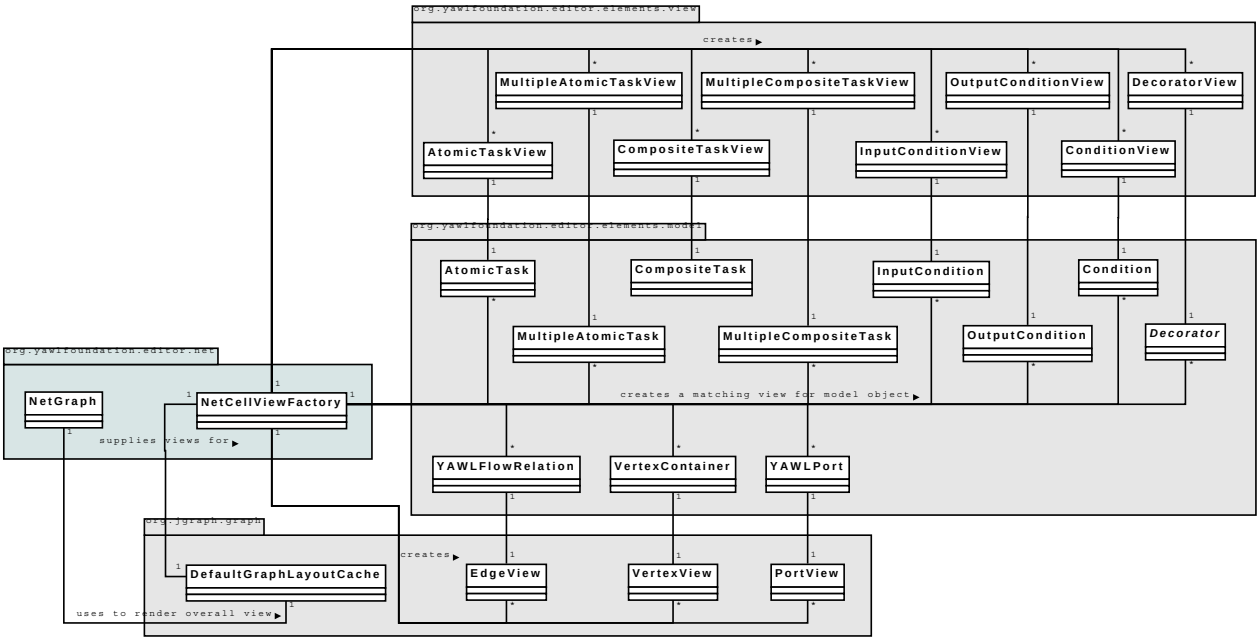


Figure 7.6: A class diagram of how model objects are associated with their views

We note from figure 7.6 that ports, flow relations and vertex containers need no special rendering, and fall back to the basic view classes supplied by JGraph. Other net elements need their own views specified, however. Note the special case of the `Decorator` and `DecoratorView` pair. Instances of `SplitDecorator` and `JoinDecorator` are ultimately what are placed on the canvas of a Net graph. However, there is significant overlap in how splits and joins graphically represent whether they are OR, XOR, or AND types. Rather than repeat the same rendering code in two classes, the one view object deals with both types of decorator.

JGraph requires that a special type of class be supplied to the view component of the graph that acts as an implementation of the factory pattern. This class is to take model objects, and return corresponding view object for them, and is ultimately invoked by the JGraph view component. We also see from this figure that we need no special view component for the actual graph of a YAWL net. The JGraph supplied `DefaultGraphLayoutCache` suffices, so long as it has an instance of `NetCellViewFactory` set as its view factory.

Active Graphing Elements

The package `org.yawlfoundation.yawl.editor.net` could be considered the heart of the editor. Specifically the two classes `NetGraph` and `NetGraphModel` from this package are where the real action for drawing YAWL nets is to be found. Figure 7.7 is a class diagram of the core net package classes and their relationship to each other. The reader should see almost immediately that all roads lead to `NetGraph`. Following close behind this class in terms of importance to editor behaviour is the class `NetGraphModel`.

A note should be made on cancellation sets, and their implementation. Every atomic task can potentially have its own cancellation set. However, only one cancellation set can be rendered on the graph at any one time. When that cancellation set is viewable, `CancellationSetModel` allows additions and removal of elements from the relevant cancellation set. We have one `CancellationSetModel` per `NetGraph`, though the actual cancellation set this model manipulates can change with user selection.

Figure 7.8 shows the cancellation set of task *a* being rendered in red. As *a* is the cancellation set triggering

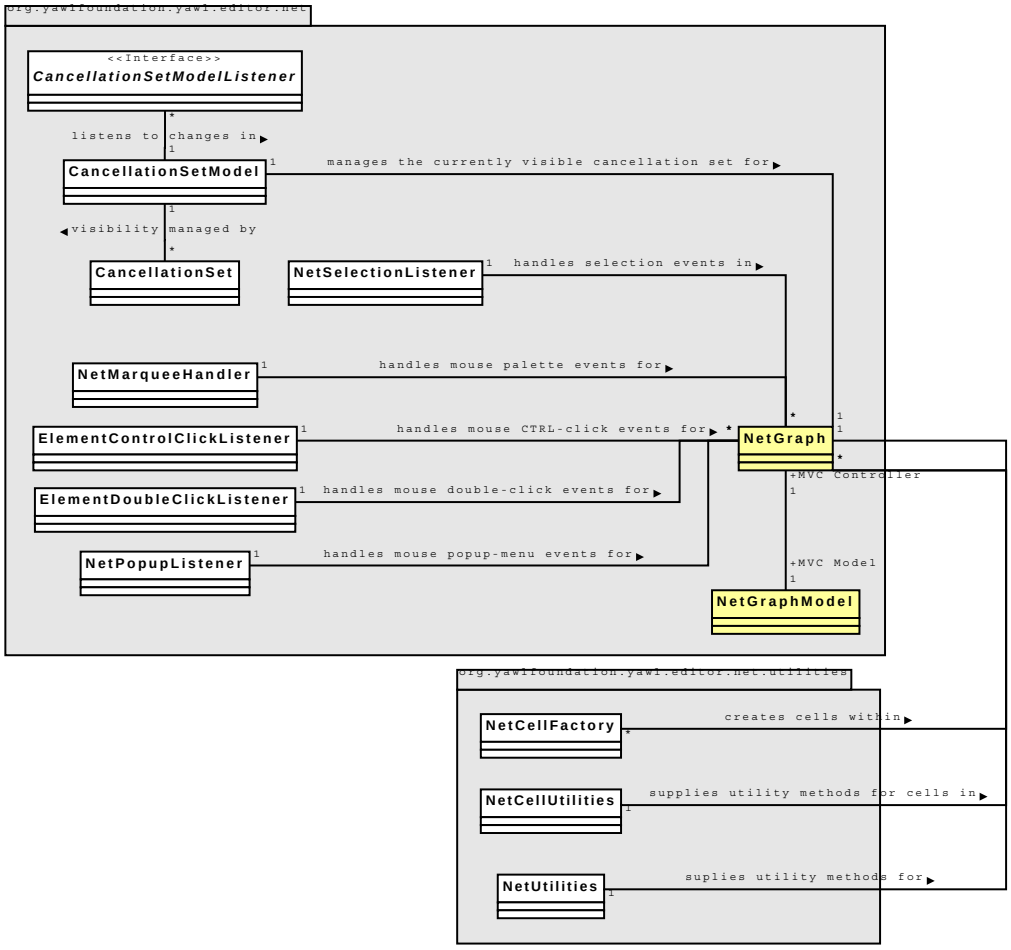


Figure 7.7: A class diagram of the core net package

task, its background is grey. Whilst the cancellation region of *a* is visible, extra tooltip buttons become enabled, allowing the user to change the composition of this cancellation set. This is a deliberate design choice that varies somewhat from the "lasso" approach described in the YAWL research publications. The lasso concept proved be difficult to implement, especially when some tasks in a rectangular region of a lasso may not be part of the cancellation set.

There are also a number of classes that register as mouse and keyboard listeners with `NetGraph` to handle various functionality triggered by these forms of input. Most of these classes are rather trivial, and quickly understandable with a perusal of the actual code. One, however, is rather complex, and deserves further discussion.

This class is `NetMarqueeHandler`, and is responsible for dealing with basic drawing on the graph, taking into account a number of factors, such as the currently selected palette button, and the type of element the mouse was hovering over when a user began using the mouse on the graph. The key behaviour of this class is to recognise and react to mouse events for placing tasks and conditions on the graph, to allow flows to be drawn between valid net elements, and to allow for the selection of a number of elements for further processing.

Earlier versions of the editor required a "flow" palette button that had to be selected to allow flows to be drawn between elements. Several users mentioned that they wished to draw flows without having to select the palette button, however. Texts on UI design[2] also suggest that removing modal behaviour, such as

[2]See "The Humane Interface" by Jef Raskin for one of the better explanation on why modal UI behaviour is to be avoided where possible.

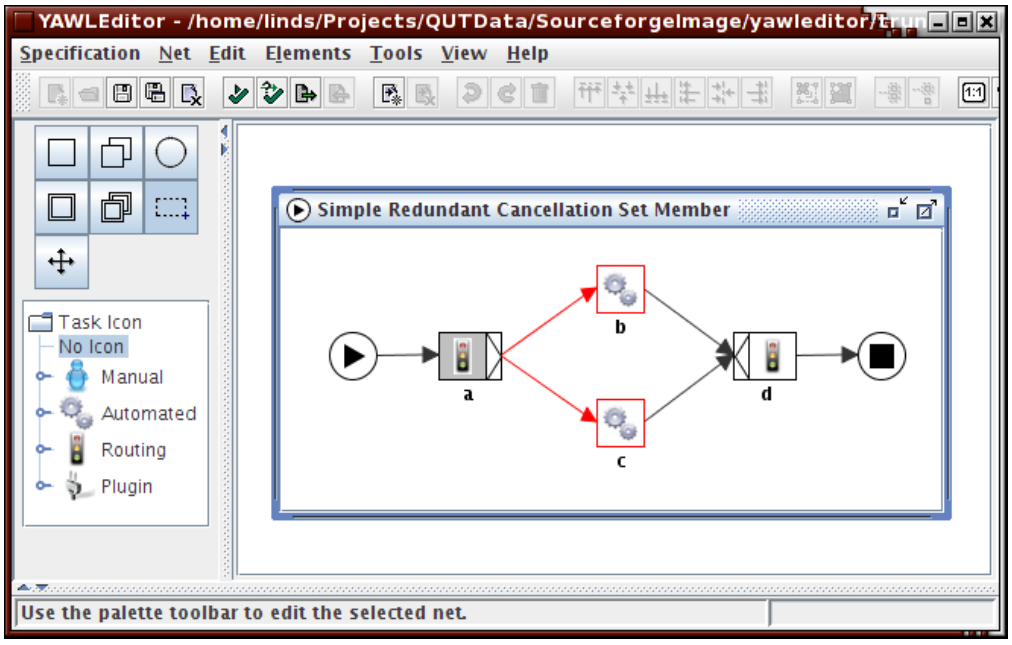


Figure 7.8: A visible cancellation set in the editor

explicitly requiring a flow palette button to be depressed for drawing a flow, be removed where possible. Thus, `NetMarqueeHandler` underwent an overhaul, and implemented a state transition machine that allowed the user to draw flows if their mouse was over a valid port, to drag an element if over the element, or drop a new element onto the graph canvas if there is nothing under the mouse. The state transition diagram used to initially design the desired change in marquee behaviour is shown in figure 7.9. It is suppled here in the hopes of allowing a quick understanding of the state transition machine built into `NetMarqueeHandler`.

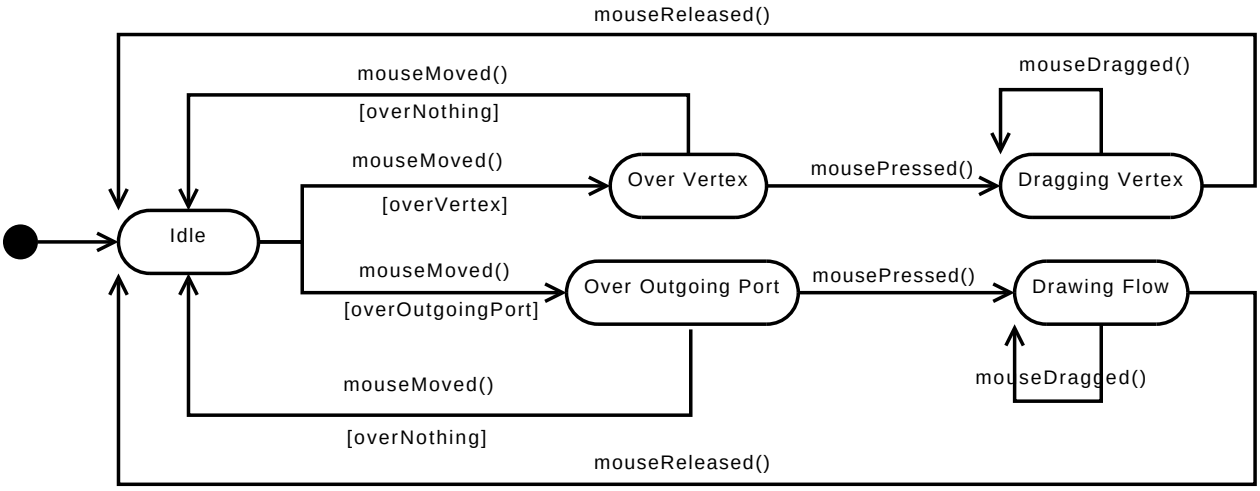


Figure 7.9: A state transition diagram for `NetMarqueeHandler`

7.2.4 Build Environment

This section describes the build environment used to create an executable editor, along with issues that developers should be aware of when having to make changes to this build environment. The editor has an

ant build script, which sits within the the source directory of the editor. All of the resources required by this script are expressed in terms of relative paths to the script. It's thus possible to decompress the source zip file anywhere, and expect to have the ant script produce runnable editor jar files. The script has a number of targets, listed below with a brief description of the result when running the script with a given target:

buildEditorComplete: Compiles, builds and packages a complete YAWLEditor.jar. This target requires the YAWL Engine's *standalone* jar in the project's *lib* directory.

buildEditorLite: Compiles, builds and packages a "lite" version of the editor, containing just core editor and JGraph code. A lite editor requires the YAWL Engine's *standalone* jar on its classpath, or will have all behaviour relying on the engine deactivated if it cannot find it.

clean: Resets the build environment to just the source, tool and library directories. This target should be run whenever the YAWL Engine's *standalone* jar is updated in the project's *lib* folder.

compile: Compiles the editor source, by compiling the core YAWLEditor.java class. This will *not* compile JUnit tests, as the editor does not reference the unit test classes. Part of the work here involves creating a compiled directory containing all resources that should be packed into a standalone jar file.

compileTestFiles: Compiles the editor's JUnit source code, and all the editor code that the JUnit code exercises.

document: Generates javadoc from the editor source code.

full: A convenience target, that invokes the clean target, then builds a lite editor, a complete editor, zipped source code and javadoc.

release: A convenience target, that builds a lite editor, a complete editor, zipped source code and javadoc ready for release.

runTests: Runs JUnit on the compiled JUnit test classes.

zipSourceCode: Generates a zipped source file image, and automatically filtered release notes and change log, all ready for minimal-fuss delivery to sourceforge.

Note the existence of the file build.properties residing in the same directory as the ant build script. The build script uses this properties file to identify current version values and dependencies the editor has on other technologies. The properties file is also used to apply filter across the editor source code, replacing placeholder text with actual text spread throughout the editor text for the current build (such a release numbers, etc.).

There are two sibling directories to the source directory that are important for generating an executable editor. These directories are lib, which contains the standalone engine jar and other required 3rd party libraries, and tools, which currently contains the JUnit jar. Build file deliverables, such as jar files or source zip files, are output to the project's distribution directory, created as a sibling to the source directory.

When it comes to actually building the editor, one can choose a "lite" or complete version of the editor. The original rationale for a "lite" editor stems from original design belief that the editor and engine should remain as decoupled as possible. A lite editor allows one to quickly ascertain whether the editor will function as intended, even without access to the engine. More will be said on coupling between the editor and engine in later sections.

7.3 Superstructure

This section describes superstructure of the editor, which uses features described in section 7.2 as the necessary operational basis for the behaviour described here. Behaviour is named "superstructure", because in a very real sense, this behaviour relies on the behaviour already described in this earlier section.

This section is broken into a number of subsections. Section 7.3.1 deals with various aspects to specification management. Section 7.3.2 describes the typical approach that has been taken to interfacing with 3rd-party software. Section 7.3.3 briefly describes Swing undo/redo functionality extensions that were needed above and beyond basic JGraph undo/redo behaviour. Section 7.3.4 describes the general approach taken to JUnit testing of the editor.

7.3.1 Specification Management

Section 7.2.3 covered editor behaviour that involved manipulating a single net's graph. A specification is composed of several nets, however. This section describes key editor behaviour for manipulating a specification. Note that the editor can currently edit only a single specification at a time. This was a design decision taken early in the life of the editor in order to achieve a usable prototype as quickly as possible. Since then, the desire for editing several specifications has been raised time and again, and will be addressed in an upcoming version. Most of the core model behaviour for specifications described here resides within the package `org.yawlfoundation.yawl.editor.specification`.

Figure 7.10 is a class diagram of the key classes in the specification package. Associations with classes in related packages are shown where it was thought appropriate.

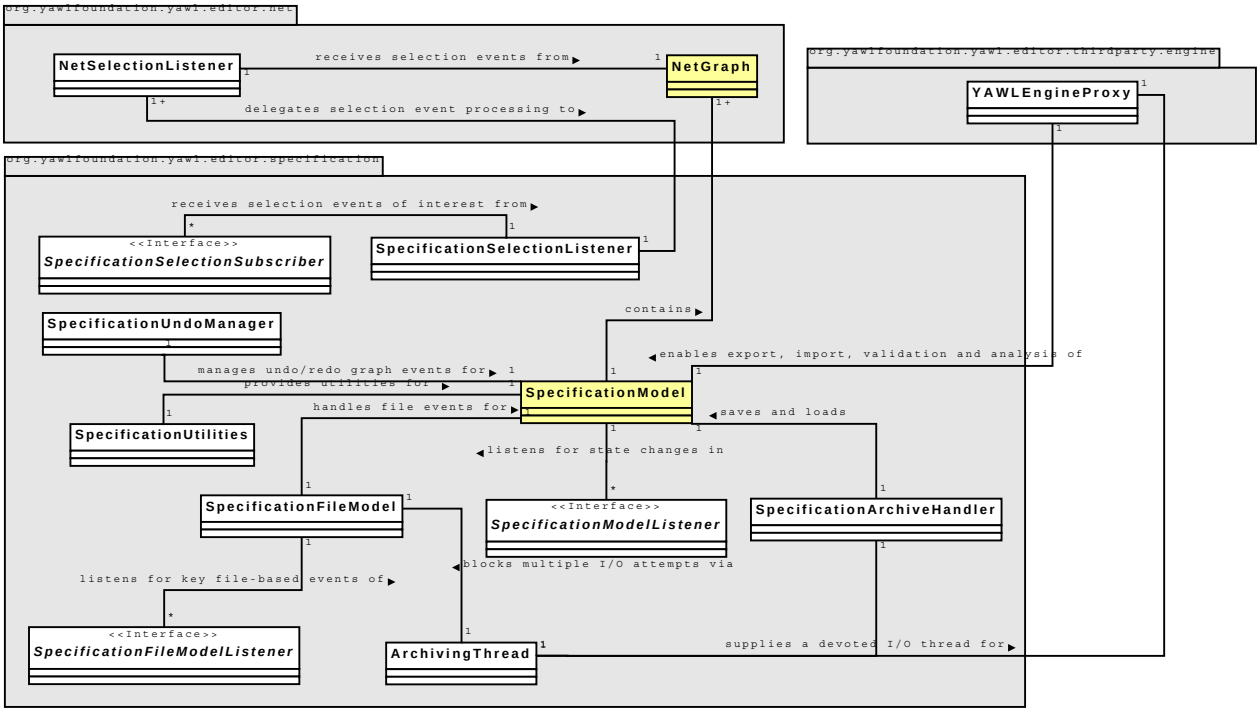


Figure 7.10: A class diagram of the core specification package

State Management

The two core specification package classes `SpecificationModel` and `SpecificationFileModel` operate as state transition machines that listeners can react to upon key state change events.

State transition in `SpecificatioModel` is rather trivial. There are two main states that it toggles between, namely `NO_NETS_EXIST` and `NETS_EXIST`. The first of these states happens only briefly on a reset of a specification model, as a valid specification must always have at least a root net. A number of events are also occasionally fired from this class, but are not states as such, just event notifications that state listeners may also care to know about. These events are `NO_NET_SELECTED`, `SOME_NET_SELECTED`, and `NET_DETAIL_CHANGED`. Classes that implement the `SpecificationModelListener` interface will be notified of these events, and react as they deem most appropriate.

State transition in the class `SpecificationFileModel` is a little more involved. A state transition diagram of it is supplied in figure 7.11.

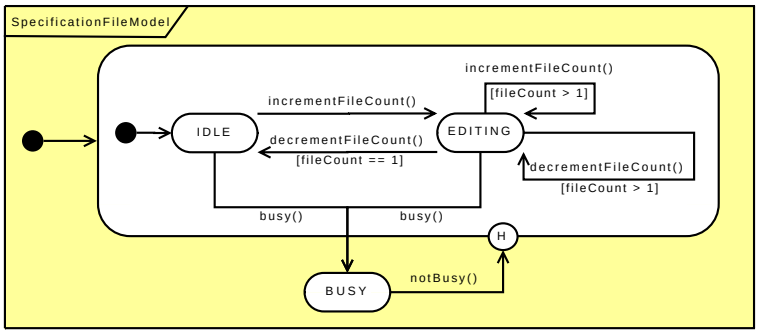


Figure 7.11: A STD describing `SpecificationFileModel` behaviour

State transition in `SpecificationFileModel` is one the few areas of the editor that caters for the possibility of having several specifications open. Again, there are two key states, being `IDLE` and `EDITING` representing an editor with no specifications open, and an editor with at least one specification open, respectively. A third state, `BUSY` can be reached at any time from either state. Once the model stops being busy, it reverts to whatever state it was previously in. Whenever `SpecificationFileModel` is busy, it is essentially blocking, allowing a devoted file I/O thread class `ArchivingThread` exclusive access to the filesystem, for the purpose of writing or reading files. Classes implementing `SpecificationFileModelListener` are expected to disallow any behaviour whilst in the busy state that might interfere with the file operations currently underway. File I/O is regulated to a dedicated thread to allow the user to continue using the editor on long file operations.

A third state management class, `SpecificationSelectionListener` has all net selection events delegated to it. In reality only a single net will ever be generating selection events, but no matter which net is currently having nodes selected, events will pass from the net selection listener to this class for processing. This class takes the currently selected set of nodes and interrogates them for patterns of selection that are of interest to listeners implementing the `SpecificationSelectionSubscriber` interface. Listeners can subscribe to be informed of a number of patterns that they may wish to react to. The current set of possible patterns tested for are listed below, and are hopefully self-explanatory:

- STATE_NO_ELEMENTS_SELECTED
- STATE_COPYABLE_ELEMENTS_SELECTED
- DELETABLE_ELEMENTS_SELECTED
- STATE_ONE_OR_MORE_ELEMENTS_SELECTED
- STATE_MORE_THAN_ONE_ELEMENT_SELECTED
- STATE_MORE_THAN_ONE_VERTEX_SELECTED
- STATE_SINGLE_TASK_SELECTED
- STATE_SINGLE_ELEMENT_SELECTED

Behaviour across the various aspects of the editor is only valid when one or more of the above patterns of selection are true. Such behaviour will implement the previously mentioned interface and subscribe for patterns where that behaviour can be applied. For example, input conditions and output conditions cannot be deleted or copied from a net. If a selection in a net selects only an input condition, the copy menu item will know that it has nothing to copy, and stay disabled until such time as a selection event includes net elements that can be copied.

Saving and Loading

The class `SpecificationArchiveHandler` is responsible for saving and loading YAWL specification files. Prior to version 2.0, an editor save file (ẏwl file) was, quite literally, an XML serialization of specification model objects that has been compressed using the popular Winzip compression algorithm. It is thus possible to open an ẏwl file with Winzip or some other compatible compression utility and interrogate the XML representing the sate of model objects in the editor at save-time. Since 2.0, these serialized files have been deprecated, and all loading and saving is done to and from ẏawl files in plain-old-XML (POX) format.

As a legacy of the pre-2.0 file serialization formats, each class that was serialized as part of the editor save/load routine uniformly stores its data members in a `HashMap` called `serializationProofAttributeMap`. Without this approach to saving and loading object attributes, the addition of a new attribute to a class would break save-file backward compatibility, stopping the editor from successfully loading older specifications that do not have a newly introduced object attribute. Note also that JGraph employs a similar technique for its core graphing elements.

Listing 7.2 shows example code to highlight the way in which state is expected to reside as attributes in these model objects. As they were ultimately to be processed by the Java classes `java.beans.XMLEncoder` and `java.beans.XMLDecoder`, all state saved and loaded needed to conform to those parts of the Java Bean standard needed by these XML utilities. Specifically, a class needs a no-argument constructor, which will be invoked when the class is initially de-serialized, and a get()/set() method pair for each attribute that they wish to serialize.

```
private HashMap serializationProofAttributeMap = new HashMap();

...

public void setSerializationProofAttributeMap(HashMap map) {
  this.serializationProofAttributeMap = map;
}

public HashMap getSerializationProofAttributeMap() {
  return this.serializationProofAttributeMap;
}

....

// example serialization-proof attribute bean methods below.

public void setSize(Dimension size) {
  serializationProofAttributeMap.put("size",size);
}

public Dimension getSize() {
  return (Dimension) serializationProofAttributeMap.get("size");
}
```

Listing 7.2: An example serialization proof attribute's get/set method pair

7.3.2 Interfacing with 3rd-party Software

Whenever the editor has to interact with the YAWL engine or resource service, an approach was adopted that allows the editor to function with or without access to this software, granting greater behaviour when it is present, and restricting functionality when it is not. These services have had a special proxy gateway constructed for them. The gateway, and associated behaviour for each are assigned a specific sub-package of `org.yawlfoundation.yawl.editor.thirdparty`.

The engine proxy will be used as a working example to describe how the editor interfaces with an external software entity. A class diagram of key classes within the third-party engine package is presented in figure 7.12. A proxy interface class is supplied for the third-party software, which defines all possible methods that the editor can invoke that will eventually result in a interaction with the third-party software. For the engine, this interface is `YAWLEngineProxyInterface`. Three classes implement this interface: the core proxy class visible to the rest of the editor, `YAWLEngineProxy`, and two classes for supplying appropriate method call response for when the third-party software is present, and when it is not. The proxy class must implement a boolean test to check for the presence of the third-party software, and delegate processing for a particular method to the most relevant proxy implementation. This framework can be viewed has drawing elements from both the proxy design pattern, and the strategy design pattern.

In this manner, the editor will continue to function in a reduced capacity if it cannot find the desired third-party software. All interfacing between the editor and the third party is now also isolated and managed from the one point, reducing coupling between the products to a minimum. Note from figure 7.12 that the available implementation of the proxy will often rely on further classes for suppling the behaviour required. Here, engine export and import behaviour is separate out into respective classes with the responsibility of mapping between engine and editor Java objects.

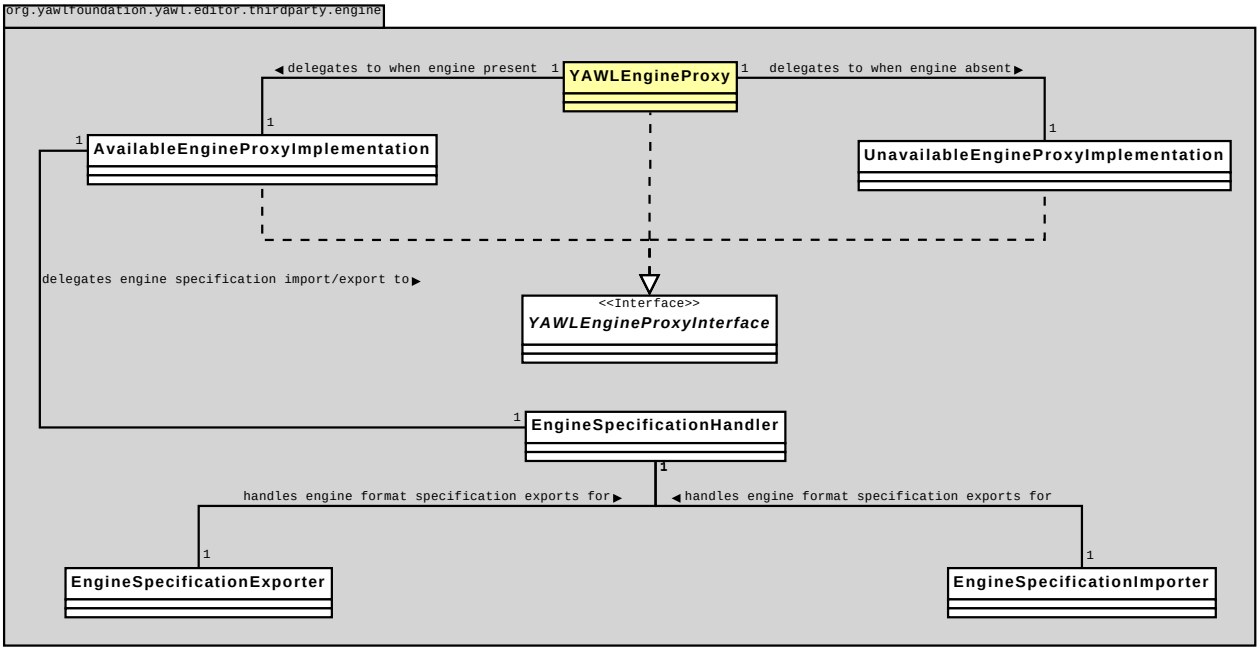


Figure 7.12: A class diagram for the third-party engine package.

7.3.3 Undo/Redo Functionality

Undo/redo functionality in an editor can be tricky to get right. Prospective developers are warned of potential gotchas to do with undo/redo generally, and certain behaviour will be pointed out to be added to the editor to better support this activity. Note that individual `NetGraph` changes are often automatically handled by JGraph without incident. However, certain changes may alter a number of nets, or are so involved

within a net that the basic JGraph undo/redo functionality has required some extension. We thus require a "somewhat more powerful that vanilla JGraph" undo/redo framework that spans all nets of a specification. Undoable event classes all reside in the package `org.yawlfoundation.yawl.editor.swing.undo`. We'll begin with the warnings.

The first warning is that, if interacting with the editor causes a visible change to one or more nets, it should be undoable. That is, if you've written code to do something a visible change, you must also write code to undo it. Sometimes, for trivial changes, the same code can handle both variants. If you want to change a node's colour then undo it, all you need do is store the old and new colour, and invoke the same colour changing code with the appropriate colour. Other times, you must write extra code that knows specifically how to undo the change, because the original change had far more long-reaching effects than just a small screen change, but both screen, and hidden changes much remain in lockstep.

The second warning is that the undo/redo code that needs to be called from the Swing-supplied framework must not have undo/redo side-effects itself. Otherwise, you may run into interesting bugs where every time you undo a change, you add something extra to be undone, that you never did in the first place. Thus, not only may you need to write specific code to undo a change, you may also occasionally have to write it in such a way as to offer two variants, one with undo/redo side-effects, and one without. Listing 7.3 shows an editor class that requires different code for the undo and redo, that must also be side-effect free.

```
public class UndoableNetAddition extends AbstractUndoableEdit {

  private static final long serialVersionUID = 1L;
  private NetGraphModel addedNet;

  public UndoableNetAddition(NetGraphModel addedNet) {
    this.addedNet = addedNet;
  }

  public void redo() {
    SpecificationModel.getInstance().addNetNotUndoable(addedNet);
    addedNet.getGraph().getFrame().setVisible(true);
  }

  public void undo() {
    addedNet.getGraph().getFrame().setVisible(false);
    SpecificationModel.getInstance().removeNetNotUndoable(addedNet);
  }
}
```

Listing 7.3: Example undoable editor class

Figure 7.13 is a class diagram, focusing on the undoable edit presented in listing 7.3. It shows how all `NetGraph` instances pass their undoable events to `SpecificationUndoManager`, which is capable of applying changes across several nets if needs be. It also points points out that `SpecificationModel` creates a new net, and passes a specification-level undoable event for its creation to the undo manager. The event code presented knows to invoke methods on `SpecificationModel` with no undo/redo side-effects for redoing or undoing the addition of a net to a specification.

The class `SpecificationUndoManager` has also been enhanced from a typical JGraph undo manager so that the programmer can:

- Briefly refuse to record undoable events. This is useful for situations where we wish to make a visible change that is controlled by a mechanism other than undo/redo. An example is the toggling of a visible cancellation set. Whether the set is visible or not is controlled by a popup menu-item for a task, and should not be reversed or re-applied via typical changes, as there is no "real" change happening to the specification here, just a visualisation to highlight aspects of existing state.

- Start and stop compounding edits. Sometimes a large change is composed of a number of small changes that by default, JGraph would make all individual undoable events. Convenience methods are supplied here to make it easy to combine a number of edits into a single edit. An example of where

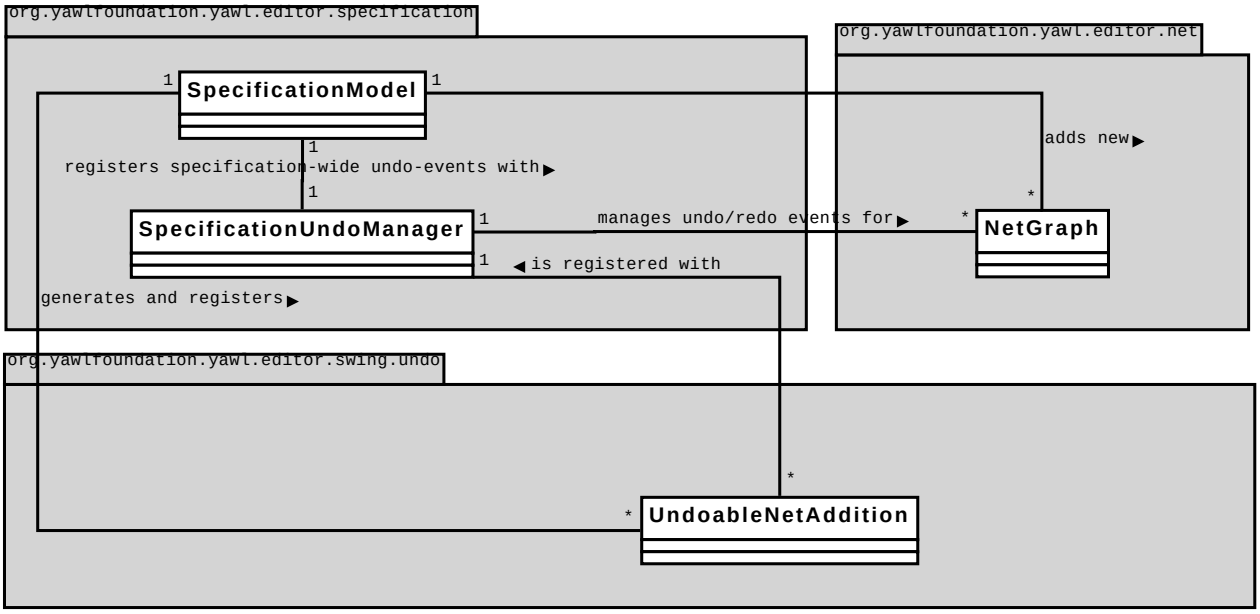


Figure 7.13: A class diagram undo/redo management of a specification.

this occurs is in `NetGraph`, where re-labeling a task is handled by deleting the old `VertexLabel` attached to a task, creating a new one, and re-balancing other elements in the task's `VertexContainer` as appropriate.

7.3.4 Unit Testing Classes

A small number of JUnit test classes have been written for the editor. Code coverage is currently not significant, given the general difficulty with attempting to apply unit-tests to a predominately GUI codebases. What unit tests there are typically lie on the border-regions of the interface between the editor and engine, as this has traditionally been where hard-to-solve compatibility issues have arisen.

The key editor JUnit class is `org.yawlfoundation.yawl.editor.TestYAWLEditor.java`. This class acts to bootstrap other JUnit classes throughout the editor. Note that a convention for JUnit test classes was followed that the reader should be aware of. A JUnit test class limits its focus to a single editor class. It takes the name of the class it tests, and prepends the text `Test` to identify the class it tests. Only JUnit classes can start with `Test`. A JUnit test class also resides in the same package as the class it tests. As described earlier, the ant build script ensures that test classes are not shipped with the final editor, and enables a way of running the JUnit tests outside of an IDE. The build script assumes the above convention in order to correctly identify the test classes.

The class `TestSpecificationEngineHandler`, sitting in `thirdparty.engine`, is often pivotal to the continued assurance that the engine and editor interact as intended. This class programatically builds a number of specifications, some valid, some deliberately flawed, and asks the available engine-side code to confirm whether the specification is flawed or not.

7.4 Future Considerations

This final section of the manual lists a number of issues that have been raised from time to time in the past, but have not yet been satisfactorily addressed, to finally free the editor of its prototype legacy. A common inhibitor for these items is time. Most of the items listed represent a significant investment in time, and in some cases, with no visible change the the editor's behaviour to show for the effort.

7.4.1 Engine and Editor Coupling

Currently, coupling between the engine and editor is still higher than preferred. In order to validate a specification, the editor builds a set of engine objects from the editor model, turns this into engine XML, and asks the engine validation methods to supply feedback on the engine XML. All of these activities rely on calls directly into core engine model objects. This has allowed for early rapid construction of an editor that can export and validate engine XML, but the necessary cost is a very tight binding between the editor model and engine model.

The ideal would be to have the editor not rely on engine objects for generating engine XML, but to generate engine compatible XML without the need to call engine library. This would reduce coupling to the engine immensely, allowing a cleaner separation between the two components than is currently possible.

7.4.2 Data Perspective Complexity

The data perspective of YAWL is complex, and requires deep XMLSchema and XQuery knowledge to successfully define non-trivial constructs. It is often cited by users as the hardest part of YAWL to understand. The editor currently exposes all of the complexity of the data perspective to users, simply because building abstractions to make it easier to wield represents a significant investment in time. Even valid combinations of scope for passing data from a net to a task and back again is not trivial. The reader is directed to `DataVariableSet` in `org.yawlfoundation.yawl.editor.data` for a decision table describing valid scope combinations.

Approaches put forward for managing data perspective complexity is to either provide better support for the full complexity of the data perspective through abstractions, or to revisit the data perspective with a view towards simplifying it. For example, converting values as they pass from the net to the task could be removed, and replaced with a simpler approach where the value passes from net to task unchanged. Any changes needed on the data needed is then forced to the task level, which is outside the scope of the engine. A simplified data perspective in the engine will result in simplified definition of that perspective in the editor.

7.4.3 Plugin Framework

Since version 1.5 of the editor, a simple plugin framework has been introduced. At the moment this framework is primitive, but it has the potential to become a potent extension to the editor. Currently, the plugin framework supports only task icons, and property files that describe extended attributes to be attached to task data variables and decompositions. Ideas for extending the plugin framework include:

- As a mechanism to allow for a library of commonly used decompositions for a particular organisation.
- In a similar vein, to allow a library of common resourcing specifications to be quickly attached to a particular task. For instance, perhaps an organisation typically automatically allocates work to its users via the roles they play in a round-robin fashion, starting the workitem immediately upon allocation. Such a resourcing decision could be pre-generated and available through the organizations plugin mechanism for quickly attaching to typical tasks in their workflows.

No doubt more behaviour will become candidates for plugin consideration if it represents a typical tightening of generic behavior to fast-track an organizations typical use of the editor.

Appendix A

newYAWL Resource Perspective Requirements Analysis

This chapter describes the design requirements for the YAWL 2.0 environment as regards comprehensive support for the resource perspective, that were derived from the newYAWL technical report [15], and Nick Russell's PhD thesis [16].

Each of these requirements has been implemented in the YAWL Resource Service and Editor. This analysis is included in this manual for historical reasons, and for consideration of some of the design decisions taken in the development of the Resource Service and relevant extension to the Editor.

A.1 Introduction

There are three key perspectives to workflow. Namely *control-flow*, *data*, and *resourcing*, representing the sequencing in which work needs to be done, the data needed to do work, and the participants needed to do that work respectively. The three perspectives each provide a view into the same underlying integrated, and ultimately inseparable whole of a workflow system.

The newYAWL technical report has identified that the resource perspective is loosely coupled to the control-flow and data perspectives (which themselves are necessarily tightly coupled). Because i) the resource perspective is so loosely coupled, and ii) the resource perspective is currently the weakest of the three perspectives of YAWL, this requirements analysis will be limited in scope to just the proposed resource perspective changes described in the newYAWL technical report. Increased support for the other two perspectives will be considered at some later stage.

Through the preparation of this analysis, it has become clear that the resource perspective changes could be handled in a two-phase approach. A number of requirements can be identified as core. They must be in place initially. The remaining requirements identified need these core requirements to be in place before they themselves can be implemented. We are therefore capable of building the resource perspective of newYAWL in at least two phases, which will go by the working titles of "core" and "non-core". Later in this document, the requirements will be allocated one of these two designations.

Note that the newYAWL technical report and its petri-nets capture key concurrency concerns between the system, and the participants and administrators. The technical report is to be viewed as the final authority on the system to be constructed. This document describes the behaviour required in natural language, which necessarily results in some loss of accuracy. Readers should view this document as an introduction to ease them into understanding the requirements which are ultimately encoded within the petri-nets of the technical report.

A.2 Requirements

This section forms the heart of the requirements analysis. To aid in streamlining software construction from this document, the requirements have been grouped into subsections, representing major existing components of YAWL. Section A.2.1 contains requirements specific to a workflow designer component. Section A.2.2 contains requirements specific to a YAWL engine component.

When new requirements are introduced, they take the following generic form comprised of three parts. Firstly, a requirement reference code of the form **REQ-###**. Secondly, a short text description in italics, and finally, a deeper discussion of the requirement is supplied in the paragraph immediately following. Below is an example:

REQ-000 : *This is an example requirement comprised of a reference code and description*

Following each requirement is a detailed discussion of the motivations that go into forming the requirement.

The requirements identified in this document are to ensure that participants interact with the work items offered by YAWL as intended by the newYAWL technical report. The requirements are described in an implementation independent manner, allowing differing implementations of YAWL to support the requirements identified as best suits each implementation.

Each requirement is classified as either *core* or *non-core*. The distinction can be rather arbitrary at times. The guiding heutistic for deciding whether a requirement is core or not is whether the requirement supplies necessary support for the concept of *interaction points* and their ability to be system or user initiated, described later. Non-core requirements can take advantage of this concept but do not necessarilly add to the fundamental behaviour of interaction points. At times, this rule has been broken if it was felt by the author that classifying a requirement as non-core would make a system, constructed to only core requirements too limiting to its users.

A.2.1 Workflow Designer Requirements

In YAWL, *atomic tasks* and *multiple-atomic tasks* both represent potential interaction points with workflow participants. If these tasks are allocated a "worklist handler decomposition" at design-time, runtime-instances of these tasks will cause the appropriate worklist handler to offer participants work items where they can receive work data and check new data back into the workflow system. For atomic tasks, there is one work item per task instance. For multiple-atomic tasks, there are a number of work items per multiple-atomic task instance.

A graphical summary of requirements for a YAWL design tool supporting the *newYAWL* resource perspective is supplied in figure A.1. The requirements are laid out in a hierarchical mind-map. Colour coding is used to isolate a group of requirements that could all belong to the same page of a wizard walk-through type screen that workflow designers could use to construct a complete resourcing definition for a task. If a group of one colour exists closer to the root of the tree, requirements from that group will be needed as prerequisites to requirement groups that appear further out. The requirements listed in figure A.1, along with related design-time requirements are described in further detail next.

REQ-001 : *Decompositions are to identify whether they need resourcing.*

Decompositions describe how a particular task in a workflow must interact with its external environment. Part of the description is what data should be passed to the external environment and what should be received. Interactions with external automated systems to YAWL typically do not allow YAWL to make

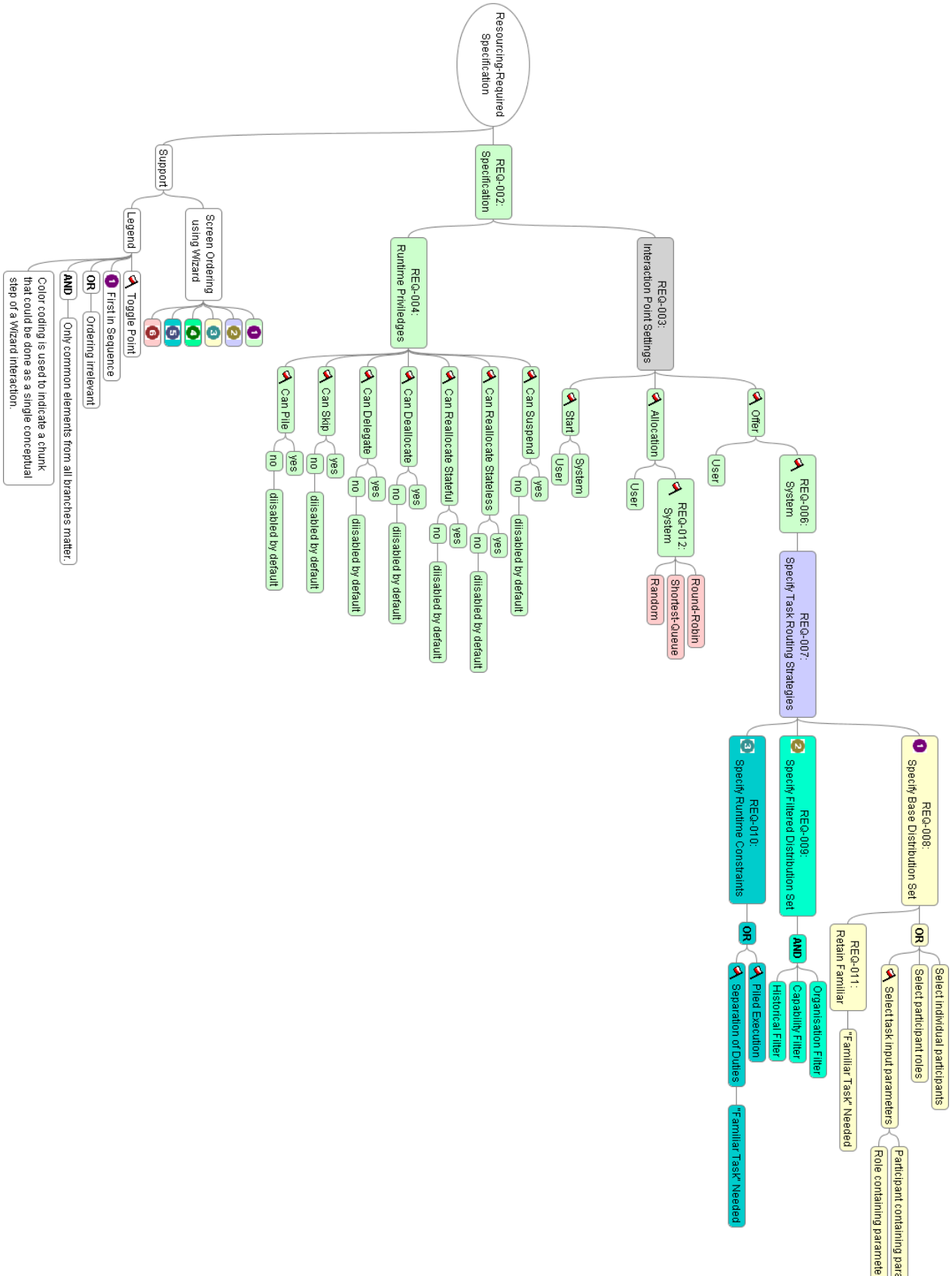


Figure A.1: Resourcing Specification in a YAWL Designer

resourcing decisions (such as say, a Web Service invocation). Sometimes, a decision also needs to be made by YAWL on how and when external participants will interact with the work item(s) of a task. Decompositions therefore need to indicate whether they need resourcing decisions made at workflow design time.

There should always be a default work-list handler decomposition offered by YAWL that is identified as needing resourcing. All decompositions identifying *automated* external interaction *will not need* resourcing decisions. All decompositions identifying *manual* external interaction *will need* resourcing decisions.

This requirement is classified as *core*.

REQ-002 : *Atomic and Multiple-Atomic Tasks with "resourcing required" decompositions will be allowed to specify necessary resourcing requirements.*

For tasks with decompositions needing resourcing, workflow designers will be able to specify resourcing requirements for those tasks at design time. Note that a number of tasks can share the same decomposition. Each task will require resourcing decisions to be made, but each task may have a different resourcing choice made. The actual resourcing decision made by the designer will thus be per task, not per decomposition.

As the resourcing decision belongs to a particular task, that decision remains fixed across all work item(s) of a multiple-atomic task, and across all work items of a task that is within a loop. It will not be possible to specify resourcing per sub-instance of a multiple-atomic task, or for each particular iteration of an atomic task work item. However, as the resourcing decision must be re-evaluated upon each work item instantiation at runtime, it will be possible for the resourcing decision to result in a number of work items for a single multiple-atomic task to be spread across a number of participants, for example.

At a high level, the workflow designer needs to be able to describe whether either users or the system is responsible for resourcing at key interaction points. They will also need to occasionally dictate what runtime privileges a user will be granted for a task. At design time, a task with resourcing required will inherit a "global" resourcing decision default that the designer must override where appropriate.

This requirement is classified as *core*.

REQ-003 : *For tasks requiring resourcing, workflow designers will be able to specify either* **system** *or* **user** *initiated behaviour for each of its* **Offer**, **Allocation**, *and* **Start** *interaction points.*

Three interaction points have been identified for where participants and the workflow system can make work-distribution choices. They are listed below in order of execution:

Offer: A work item can be offered to a participant. This does not imply that the participant *should* undertake that work, merely that they *could*. A number of participants could be offered the same piece of work, for example.

Allocation: Allocating a work item to a participant means that participant has been committed (willingly or not) to doing that work item some time in the future. This participant no longer *could* do the work, but rather *should* do it. The work item is withdrawn from the offered queues of other participants.

Start: A work item goes from allocated to started once work has begun. The participant that once *should* have been doing the work has now started doing so.

For relevant tasks, workflow designers will be able to specify how each interaction point should be handled in a running workflow by choosing whether that interaction point will be *system initiated* or *user initiated*. System-initiated offers and allocations require further decisions to be made by the workflow designer to guide the running workflow system in its choices.

This requirement is classified as *core*.

REQ-004 : *For tasks requiring resourcing, a workflow designer must be able grant or deny a number of user-task*

privileges for the task's work item(s).

A workflow designer will be supplied a number of user-task runtime-privileges, that can be either enabled or disabled for the resourcing-required task. These privileges are so-named because they will only be enabled for a (user, task) combination. The privileges, and their runtime effect when enabled are described below:

Can Suspend: When enabled, participants who've started work items of the task will be allowed to suspend the execution of the task.

Can Reallocate Stateless: When enabled, participants who've started work items of the task will be allowed to reallocate the work items to other users. The process of re-allocation will revert the work item to an initial starting state.

Can Reallocate Stateful: When enabled, participants who've started work items of the task will be allowed to reallocate those work items to other users. The process or re-allocation will preserve the current state of the work item.

Can Deallocate: When enabled, participants allocated work items of the task will be allowed to de-allocate themselves from the work items, causing a reallocation of the work item.

Can Delegate: When enabled, participants allocated work items of the task will be allowed to reallocate the work items to other users, so long as they have not yet been started.

Can Skip: When enabled, participants allocated work items of the task will be allowed to skip (or finish without starting) the work item. Therefore, this privilege will only be grantable on tasks that supply default values for all of their output parameters.

Can Pile: When enabled, a participant will be allowed to pile the execution of all work items of this task upon themselves. Instead of the workflow system building a distribution set of viable participants first, it will directly allocate and automatically start new work items for the task against the relevant participant.

To avoid a combinatorial explosion within a running engine in relation to setting individual permissions against combinations of users and tasks, we also need a switch per enabled-privilege. The switch will indicate whether all users that are allocated the task are granted the privilege by default or not. An administrator will still be free to override the default, but we expect the default to cover the desired behaviour of most users involved in the given workflow.

The priviliges discussed above can be viewed as *non-core*. Administrative functions described later will be capable of the short-circuiting of work items through to alternate worklist states, which can act as a work-around until such time as non-core requirements can be constructed for many of the privileges here. This will place extra burden on aadministrators until such the non-core requirements are constructed. Those privileges that have no administrative fall-back behaviour do not contribute to supporting interaction point behaviour, but will result in the loss of some pattern support.

REQ-005 : *To support the* **Can Skip** *user-task privilege, tasks will be able to specify default values for their output parameters.*

The workflow designer will be allowed to specify an optional static value for any of its output parameters. If the designer chooses optional, they will also be asked by the design tool to supply a static "default" value for the optional parameter. This optional value will be used to create "completed" data when the work item is automatically completed upon a user-request to skip that work item. Note that all of a task's output parameters must have optional values specified in order for it to be validly skipped at runtime.

This requirement is classified as *non-core*.

REQ-006 : *For tasks requiring resourcing, a default decision of* **user offer**, **user allocation**, **user start**, *will apply for a task's interaction points. No privileges will be granted by default.*

All tasks that require resourcing need a default resourcing decision that a task inherits initially, and that a workflow designer can choose to override as necessary. Defaulting is required for interaction points and runtime privileges.

For the three interaction points described earlier, the default will be *user* for each interaction point. This will result in all decisions on resourcing to be manually handled at runtime within a YAWL engine. For the user-task privileges, all privileges will default to being disabled, giving users minimal freedom in their ability to manipulate work items of the task at runtime.

This requirement is classified as *core*.

REQ-007 : *Workflow designer to specify how a System-Initiated Offer will choose participants.*

When the workflow designer identifies that offers should be system-initiated (regardless of what choices they make for the allocation and starting points), the workflow designer will also be required to specify a distribution strategy for the offer, with ultimately tells the running workflow system how to generate a list of participants to which a work item should be offered.

Initially this will involve having the design tool ask the workflow designer to specifying a number of *task routing strategies*, and optionally, to specify whether the workflow system will retain a "familiar participant" for allocating a work item to. Both options are described in a number of requirements that follow.

This requirement is classified as *core*.

REQ-008 : *Workflow Designer to be able to specify task routing strategies for a resourcing-required task*

When a workflow designer has chosen to specify *task routing strategies* after having specified that offers are to be system-initiated, the design tool will require the following sequence of steps to be followed by the workflow designer:

1. Specification of a base distribution set,
2. Specification of filters for the base distribution set using organisation, capability and historical data,
3. Specification of a number of runtime constraints for this filtered distribution set.

Each step is further described in requirements that follow.

This requirement is classified as *core*.

REQ-009 : *Workflow Designer to be able to specify a base distribution set of participants.*

A workflow designer begins building a task routing strategy by defining a base distribution set of participants. The designer will be able to do this by selecting participants through a number of mechanisms. The set of participants represented by these selections will be used at runtime to generate a base distribution set of participants to whom work items of the relevant task will be offered. The three mechanisms defined for adding participants to this base distribution set are:

- choose zero or more of the *participants* registered as users to the workflow system,
- choose zero or more of the *roles* defined by the workflow system,

- choose zero or more of the *input parameters* to the relevant task, and specify for each of these parameters whether the value of the parameter will contain a set of participants, or a set of roles at runtime.

This requirement needs the workflow designer and engine to communicate, where the engine supplies a full set of participants, and a full set of roles to the workflow designer. For reasons more fully in the following requirement, the identification of roles and participants must be done in a data storage independent manner to allow a range of technologies to pass resourcing data between the workflow engine and the design tool.

This requirement is classified as *core*.

REQ-010 : *Workflow Designer to be able to specify filters for a base distribution set of participants using organisation, capability and historical data.*

Once a workflow designer has defined a base distribution set of participants, they can then specify a number of filters that are to apply to the participant set. Three different categories of filter have been defined below:

- Organisation Filter
- Capability Filter
- History Filter

At runtime, all categories of filter will have access to the base distribution set of participants, plus any extra data relevant to their category (detail of which is described in the *newYAWL* technical report). Regardless of category, the filters constructed will generate a subset of the base distribution set of participants. Only those participants of the base distribution set that are valid members within *all* of the filters chosen will be included within the filtered distribution set of participants.

The workflow designer will be offered all filters registered with a running YAWL engine. They may then choose to enable a number of these filters, and supply data for whatever parameters each filter requires to enact its algorithm at runtime within the engine.

This requirement needs the design tool and engine to communicate, where the engine supplies a the design tool the full set of filters and necessary parameters. It also requires the engine specification to be flexible enough to identify a filter and the parameters values for it. Filters are to be built as plugins into a running engine, and cannot thus be hard-coded into a workflow specification.

Initially, one filter per category will be constructed to offer a "typical" simple filter for a category. These filters will be backed by a relational database that the filters will query. However, We expect alternate filters may need to draw their resourcing data from alternate technologies to relational databases. LDAP servers are often quoted as a typical example.

Note that the backing technology used to house the data needed for these filters will also be the same technology that stores participants and roles. The engine processing that reports participant and role lists to the workflow designer should allow for alternate technologies to relational databases to supply this data, but will be released initially with participants and roles being stored within a default relational database that comes with the engine. It is expected that a particular data storage technology will require its own resourcing plugin to a YAWL engine, and at the very least must offer the ability to retrieve all participants and rolls registered within its data storage technology.

An alternative to the above approach would be to construct a language, catering a large range of possible filtering strategies. The design and implementation of such a language with no clear understanding typical or desired filters is considered to be a poor investment of programming effort at this stage.

This requirement is classified as *non-core*. A rudimentary degree of filtering can be achieved by allocating participants to rolls that encode simple organisation or capability detail. It should be noted, however that full pattern support for organisation, capability and history based patterns cannot be claimed until this requirement is implemented.

REQ-011 : *Workflow Designer to be able to specify of a number of runtime constraints for this filtered distribution*

set.

A workflow designer needs to be able to define two runtime constraints for a filtered distribution set. One constraint, *separation of duties* must be enforced on a filtered base distribution set of participants. The other, *piled execution*, allows the filtered distribution set to be ignored under certain circumstances at runtime. Both runtime constraints are boolean. However, *separation of duties* also requires a "familiar task" to be chosen whenever it is active. The constraints, and the effect they may exhibit at runtime, are described below:

Retain Familiar: When this constraint is enabled, it must also have a "familiar task" chosen to be compared against. This constraint attempts to ensure that at run-time, the last participant that completed a work item of this familiar task within the same process instance will be also allocated work items spawned off the relevant resource-requiring task.

Separation of Duties: When this constraint is enabled, it must also have a "familiar task" chosen to be compared against. This constraint ensures that the filtered distribution set will not include any participant who completed work item(s) of the "familiar task" in the current case.

Piled Execution: Enabling this constraint will allow a participant to choose at runtime for the task, to receive all work items for the task across all cases of the workflow specification, thereby overriding the distribution set of participants specified for that task. Note that at run-time only one user can be piling the execution of a task's work items at any one time.

For the *retain familiar* constraint, The relevant "familiar participant work item" must also have completed before the resourcing required work item for this runtime constraint to be enacted. Thus, only resourcing-required tasks within the that lie on a backward path to the starting net's input condition from the resource-requiring task will be selectable as the familiar participant task at design time. This rule also holds for *separation of duties* familiar tasks.

For the constraints requireing familiar tasks, note that concurrency could still result in the specified familiar task not having a completed work item being ready for determining the farmiliar participant. In such circumstances, the engine should fall back to typical behaviour where it generates a distribution set of participants to offer the work-item to.

The *separation of duties* and *retain familiar* constraints are to be considered *core*. The *piled execution* is to be considered *non-core*. Piled execution is an alternative to the *interaction point* behaviour described to date. It's exclusion will deny pattern support, but it can be added into a system once the base interaction point behaviour has been established.

REQ-012 : *Workflow designer to choose System-Initiated Allocation mechanism.*

If the set of participants offered a work item contains only a single participant, the running workflow system does not need to choose between alternatives, and may allocate the work item directly when system-initiated allocation is being used. However, no guarantee can be made at design-time that an offer will be limited to a single participant at runtime. A design-time choice needs to be made to tell the YAWL system how it must choose a single participant from a number of possible alternatives for allocation. Ultimately, so long as an algorithm takes a participant set, and returns only one of those participants, it could be considered a valid automated allocation mechanism.

Initially, three mechanisms will be constructed. A workflow designer may chose from these three mechanisms:

Round-Robin: From the identified participants, choose the one who recently completed a work item for the task. In the event that several users are recorded as being equally least-recent, choose one of these randomly.

Shortest-Queue: From the identified participants, choose the one whose sum of allocated and started work item queue numbers is shortest. This will be used as the default choice.

Random: Choose a participant from the collection identified randomly.

The mechanisms will be registered with the workflow engine and communicated to the workflow designer when they come to choose an appropriate mechanism for system-initiated allocation. As new mechanisms are programmed and registered with a running engine, they will be subsequently made available to the workflow designer to choose from as well.

This requirement will need a workflow specification passed from the workflow designer to a newYAWL engine to be sufficiently flexible to allow new mechanisms to be identified within the workflow specification without requiring changes to the specification structure.

This requirement is classified as *core*.

REQ-013 : *Resourcing Decisions made must be encoded within the task definition of a YAWL specification.*

The specification generated by a workflow design tool for consumption by a newYAWL resourcing-enabled system must encode the resourcing-required choices made by a workflow designer that have been described in this section. The requirements listed below require special effort to ensure that as new options are made available by an engine for key decisions, the specification does not require structural changes in order to have them identified:

- Workflow Designer to be able to specify filters for a base distribution set of participants using organisation, capability and historical data.
- Workflow designer to choose System-Initiated Allocation mechanism.

This requirement is classified as *core* only so-far as is encodes other *core* requirements described to date for enactment within a newYAWL resourcing-enabled system.

A.2.2 Workflow System Requirements

One of the quintessential elements of a YAWL workflow is the work item, representing a piece of work that an external participant must do in order to progress a workflow. It is through a work item that an external participant and the workflow system must interact. Work items are offered to a participant via the participant's view into a *worklist handler* component. Besides the many and varied ways of classifying users of a newYAWL system that have been identified earlier, users can also act as *administrators* to YAWL. Administrators have an extra responsibility of manually coordinating the placement of work with participants where appropriate.

We therefore have two basic workflow system roles that users may play in newYAWL, *participants* and *administrators*. A workflow system role is not to be confused with organisation roles described earlier. What role(s) a user plays *within an organisation* matters only so far as they can be used by the workflow system to short-list possible participants that may do a work item. Anybody who does work through YAWL plays the workflow system role of *participant*, regardless of how they are shortlisted for being offered work items to, be that through an organisational role, or some other mechanism.

Figure A.2 is a mind-map showing a hierarchical view of screens that must be supported to allow both administrators and participants to interact with YAWL in order to support the resource perspective changes of the newYAWL technical report. The requirements that describe user interfacing have also been marked where appropriate.

This section is broken into three sub-sections. Section A.2.2 describes requirements for YAWL workflow administrators. Section A.2.2 describes requirements for participants doing non-administration duties via the worklist handler. Finally, section A.2.2 describes automated requirements that are independent of those described in the previous sections.

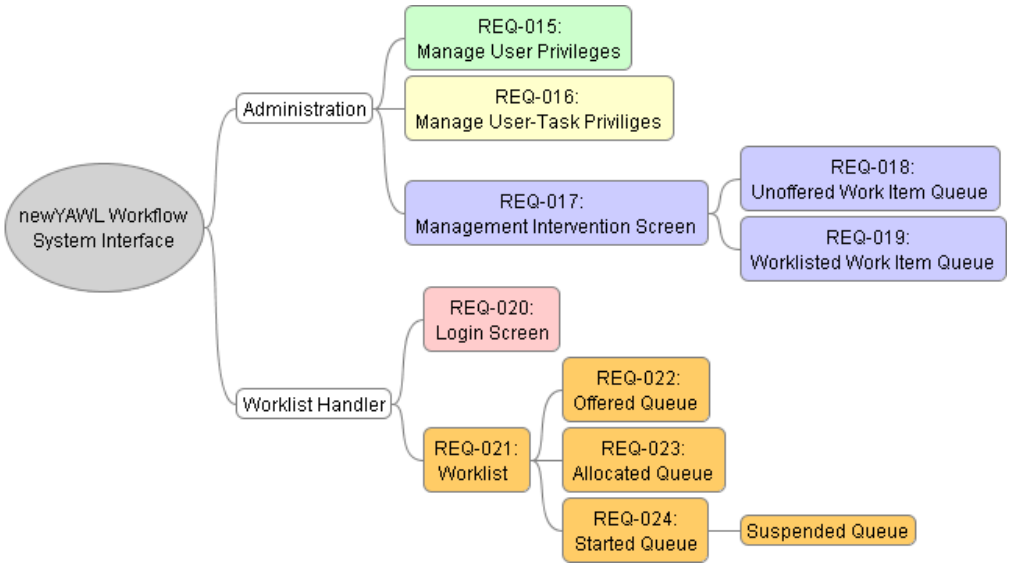


Figure A.2: A Screen Hierarchy of Resourcing support for a newYAWL System

Administration Requirements

This section describes requirements for user interfacing that administrators will need to support the resource perspective of newYAWL.

REQ-014 : *YAWL Administrator to be able to manage user privileges*

YAWL administrators will be supplied functionality to select a number of workflow participants and change their user privileges. The default setting for all privileges is to be disabled. The following privileges, when enabled for a participant, will apply across all cases that the participant can be involved in:

Can Choose Work Item to Start: When enabled, this privilege allows a participant to choose which work item in their allocated queue they will be allowed to start. When this privilege is disabled, the participant will not be able to select which work item to start. Instead, the work item at the top of the allocated queue of the participant will be started when the participant triggers a start action.

Can Start Concurrent Work Items: When enabled, this privilege allows the participant to place new work items in their started queue before they complete existing work items. Note that this does not stop an administrator or the system from placing extra work items on a user's started queue when they already have started work items. When disabled, a participant will only be able to add a work item to their started queue when the queue is empty.

Can Reorder Work Items: When enabled, this privilege allows the participant to reorder the work items in their allocated queue. When disabled, the work items will be added to the end of the queue, meaning that the longer a work item has remained in the allocated queue, the higher to the top of the queue it will be.

Can View All Offered Work Items: When enabled, this privilege allows the participant to view the offered queues of all other workflow participants.

Can View All Allocated Work Items: When enabled, this privilege allows the participant to view the allocated queues of all other workflow participants.

Can View All Executing Work Items: When enabled, this privilege allows the participant to view the started and suspended queues of all other workflow participants.

Can Chain Work Item Execution: When enabled, this privilege allows the participant chain a number work items together so that so that a new task in the sequence of work will be allocated to them and automatically started.

There is some argument that all of the above user privileges should be viewed as *non-core*, given that they are all dependent on the concept of interaction point behaviour, but do not add to that behaviour. However, the top three privileges are viewed by the author as being highly desirable ways of toggling a particular participant's way of working that we could reasonably expect to be desired by workflow systems. Therefore, the top three requirements are to be considered *core*, and the remaining "view all" privileges as *non-core*.

REQ-015 : *YAWL Administrator to be able to override default user-task privileges*

User-task privileges are initially configured through settings supplied in the workflow specification as described by a workflow designer. However, an administrator may be required to vary user-task privileges away from the defaults set in the workflow specification.

Administrators will be supplied a user-task privilege maintenance screen. The aid in navigating a large number of possible tasks, A tree hierarchy of specifications will be supplied. Each specification lodged with the workflow system will be listed and will be openable to reveal individual nets. Each of these nets can be further opened to show all tasks within that net that have resourcing requirements. These tasks can then be further opened to show each user-task privilege, along with the default setting for the privilege. The privilege can then be further opened to reveal a list of participants that explicitly go against the default setting. The administrator will then be allowed to add or remove users from this list as necessary.

This requirement is classified as *non-core*.

REQ-016 : *YAWL Administrator to be allowed management intervention via an* **Unoffered** *queue and a* **Worklisted** *queue*

YAWL Administrators will require a management intervention screen for managing work items in various states of their lifecycle. From the perspective of a YAWL Administrator, work items are either **Unoffered**, or they are one of the worklist states described later (**Offered**, **Allocated**, **Started** or **Suspended**). Given the fundamentally different nature of the actions that can be performed between unoffered work items and those that have begun appearing on the worklists of participants, the management intervention screen will separate work items into two separate queues. One queue will contain all **Unoffered** work items. The other will contain the remaining worklisted work items.

The actions that can be applied to both queues are described in the succeeding requirements. The petri-net for management intervention is figure 33 of the newYAWL technical report.

This requirement is classified as *core*.

REQ-017 : *YAWL Administrator to be able to enact administrator-initiated offers, allocations and starting of* **Unoffered** *work items in the management intervention screen*

Those work items whose resource-requirements indicate user-initiated offers, or that have failed to be successfully distributed to a set of participants will appear on the Administrator management intervention screen with a status of **Unallocated**. The administrator must be able to select a number of these work items and for them, specify a number of participants. Once the participants have been identified, the administrator will be able to enact one of three possible actions on the work item selected:

Offer: Selecting this action will result in the relevant work item being offered to the selected users. In this manner, user-initiated offers are implemented.

Allocate: This action will only be possible when a single participant has been selected. Enacting this action will directly allocate all selected work items to the chosen participant.

Start: This action will only be possible when a single participant has been selected. Enacting this action will directly allocate all selected work items to the participant and automatically place the work items on the participant's started queue.

This requirement is classified as *core*.

REQ-018 : *YAWL Administrator to be able to escalate worklisted work items from the management interventions screen*

The worklisted queue of the management intervention screen will display, by default, all work items that are offered, allocated, started and suspended. To ease the administrative burden of locating desired work items from a potentially very large list, the administrator's screen will initially display all work items currently worklisted, along with whoever has been assigned the work item, and the work item's current state. They may then choose to filter the work items on a per-participant basis, and a per-status basis. When an administrator chooses both a participant, and a status, the filter will show only those work items of the required state that have been assigned to to the selected participant.

From the queue, the administrator will be able to escalated a number of selected work items to a potentially different set of participants. They will be able to select a number of work items, as well as a number of participants that they may then enact one of the following actions upon:

Escalate Offered, Allocated, Started or Suspended to Offered: All selected work items will be withdrawn from the offered, allocated started and suspended queues that they were initially assigned to. The work items will then be added to the offered queues of the newly selected participants. Started and suspended work items will have their state reset to an initial state.

Escalate Allocated, Started or Suspended to Allocated: This action will only be possible when a single participant has been selected. All selected work items will be withdrawn from the worklists of the participant to whom the work items were assigned. The work items will then be added to the allocated queue of the new participant. Started and suspended work items will have their state reset to an initial state.

Escalate Started to Started: This action will only be possible when a single participant has been selected. All selected work items will be withdrawn from the started or suspended queues of the participants to whom the work item was assigned. The work items will then be added to the started queue of the selected participant, preserving state in the transfer. *Not currently in report.*

This requirement is classified as *core*.

Work List Handler Requirements

A participant's worklist handler is their interface into the various work items that they must interact with in order to contribute to the execution of an overall workflow instance, or case. Where a work item sits within the worklist of participants depends ultimately on the state of the work item. Figure A.3 is a UML Statechart of the lifecycle of the newYAWL work item. Figure A.4 informally describes each state that appears in figure A.3. Note that the state chart does not cover all possible state transitions. It's primary focus is to describe how the work item changes from the perspective of workflow participants.

Transitions between states in figure A.3 take a general form of "*number. transition name*", referring the figure number in the newYAWL technical report and the actual petri-net transition within that figure that causes a work item state change. One thing that immediately becomes obvious is that a work item can only go through a small number of possible states in its lifecycle, but there are many possible valid transitions between those states.

Figure A.3: Work-Item State Chart

State	Description
Initial state	The work item exists, but has not yet had a possible set of participants identified to be distributed to.
Distributed	A distribution set of possible workers exists for the work item.
Offered	The distribution set of possible participants have been offered the work item to do.
Allocated	One of the possible participants that could do the work item's work has been allocated the work to do.
Started	A participant, previously allocated a work item, has begun its work.
Suspended	A participant has suspended a work item they started earlier.
Final state	The work item is complete.

Figure A.4: State for the Work-Item State Chart

The requirements in this section spell out how a work list handler for newYAWL must interact with participants in managing work items that participants may work on.

REQ-019 : *Participants are to have login and preferences screens that support specifying chained and piled execution*

Upon login, participants will be offered a typical username and password interface. At this time, participants will also be able to toggle whether they wish to pile execution for those tasks have they have been granted the **Can Pile** privilege. Only one logged-in participant may pile the execution of a task's work items. Only those tasks that do not have a logged-in participant piling their work items will be offered to the participant for toggling the piling of. The default for the toggle shall be to not pile execution.

If the participant has also been granted the **Can Chain Work Item Execution** privilege by an administrator, the login screen will also allow the participant to toggle whether they wish to chain execution of work items. The default for the toggle shall be to not chain execution.

A preferences screen should also be provided to the logged in user, allowing them at any stage whilst they are looged in to toggle these execution mode preferences.

The petri-net for participant login is figure 71 of the newYAWL technical report.

A participan login screen is considered *core*. However, this requirement is classified as *non-core* in that chaining and piling are not core requirements.

REQ-020 : *Participants are to be offered separate queues on their work list handler for work items that are within the states of* **Offered**, **Allocated** *and* **Started**.

Looking at figure A.3, we see that typical workflow participants can only interact with work items when the work items are in the states of *Offered*, *Allocated*, *Started* and *Suspended*. Collectively, we label these states the *worklist states*, indicating that whilst within one of these states, participants may interact with a work item in certain ways. Each of these states dictate a limited number of valid interactions that the participant may perform whilst a work item is in a particular state. Interactions per state are described in later requirements.

Participants will be offered a worklist handler interface that clearly separates work items into their respective states. For screen real-estate concerns, it does not seem necessary for all queues to be visible to the participant simultaneously. A tabbed interface may be most appropriate. However, a special case may be that of the *Started* and *Suspended* queues. There is some interface pressure for these two queues to appear on the same screen. It is expected that the state of both the started and suspended work item queues for a user will influence which suspended work they should resume once the participant completes a work item.

A word of warning is offered here on web-based interfaces supporting this requirement. Web-techologies assume a client-pull model (the participant asks the sever for work updates with an explicit screen refressh action made by the participant). It is possible to construct a server-push interaction where the workflow system pushes new or updated work items out to a participant's web browser, but this requires the use of

AJAX technologies, which are a immature at the time of writing this document, and likely to consume a significant amount of effort to get working across the most popular Web browsers. Client-pull will involve less programming effort, but will not be as user-friendly as server-push.

This requirement is classified as *core*.

REQ-021 : *Workflow Participant to be able to enact self-allocation from their* **Offered** *queue, as well as view all offered work items when appropriate.*

For those work items whose resourcing requirements indicate *user-initiated allocation*, the participant must have a mechanism for self-allocating work items that appear in their offered work item queue. The participant will be able to select a number of the work items on this queue and enact an *allocate* action upon them.

For each of the work items that the participant allocates in this manner, if a work item is also *user-initiated start*, the action will move the work item from their offered queue to their allocated queue and withdraw the work item from the offered queues of other participants. If the work item is instead flagged for *system-initiated start*, the action will move the work item from the participant's offered queue directly to their started queue and withdraw the work item from the offered queues of other participants.

The petri-net requirement for doing this user-initiated allocation is described in the *process selection request* transition in figure 46 of the newYAWL technical report.

If the participant has the **Can View All Offered Work Items** privilege set, the participant will also be supplied a toggling mechanism, allowing them to view (but not otherwise manipulate) all other offered work items in the entire system. So as not to interfere with their own offered work items, these other offered work items will appear in a visually distinct queue on the same screen as the participant's own offered work items. An indicative screenshot of this requirement is supplied in figure A.5.

This requirement is classified as *core*. However, the ability to view all other work items of the same state is *non-core*.

REQ-022 : *Workflow Participant to be able to enact self-start, delegation, self-deallocation and toggle viewing all work items from the* **Allocated** *queue when appropriate.*

For those work items whose resource-requirements indicate *user-initiated start*, the participant will have a number of allocated work items available to start. A *start* action will be made available to allow the participant to move a work item from their allocated queue to their started queue.

Unless the user privilege **Can Start Concurrent Work Items** is enabled for a participant, they will not be able to activate the start action until their started work item queue is empty. If the user privilege **Can Choose Work Item to Start** is enabled, the participant will first need to select a work item to enact the start action upon. When this privilege is disabled, the top-most allocated work item will be started. The petri-net requirement for doing this user-initiated start is captured in the *start* transition in figure 48 of the newYAWL technical report.

With certain privileges enabled, participants may also be able select a work item from their allocated queue and enact one of the following actions:

Delegate to another: If the combination of participant and work item has the **Can Delegate** privilege enabled, the participant may chose some other participant to delegate the work item to. The effect of this will be to move a work item from the allocated queue of this participant to the allocated queue of the delegate. Note that only those participants that may be validly offered this work item (i.e., those listed in the distribution set of the work item'm task) will be offered as possible delegates. The petri-net transition for this is named *route delegation* in figure 51 of the newYAWL technical report.

Deallocate themselves: If the combination of participant and work item has the **Can Deallocate** privilege enabled, the participant may chose to deallocate themselves from the work item. The effect of this will be to remove the work item from this queue, and place it on the unallocated queue of the admin-

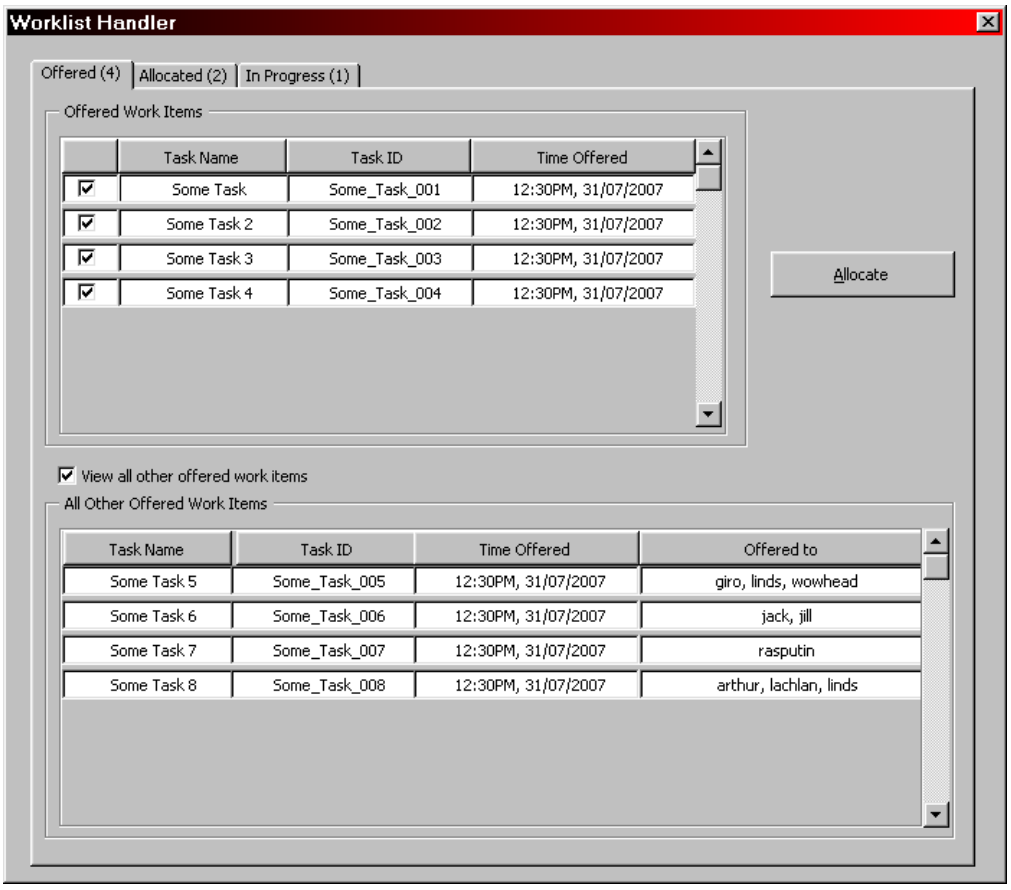


Figure A.5: A possible offered queue implementation interface

istrator management intervention screen discussed earlier. The petri-net transition for this is *process deallocation* in figure 52 of the newYAWL technical report.

Reorder work item: If the participant has the **Can Reorder Work Items** privilege set, the participant will be allowed to select a work item in the allocated queue and move that work item up or down in their work queue. Work items closer to the top of the queue are expected to be started before work items further down in the queue, which is especially true when the **Can Choose Work Item to Start** privilege is disabled for the combination of participant and work item.

Skip a work item: if the combination of participant and work item has the **Can Skip** runtime privilege enabled, they will be allowed to select a work item from the allocated queue and skip it, automatically completing that work item, using default values of data that the work item must output to subsequent work in the case.

Toggle view all work items: If the participant has the **Can View All Allocated Work Items** privilege set, the participant will be supplied a toggling mechanism, allowing them to view (but not otherwise manipulate) all other allocated work items in the entire system. So as not to interfere with their own allocated work items, these other allocated work items will appear in a visually distinct queue on the same screen as the participant's own allocated work items.

An indicative screenshot of this requirement is supplied in figure A.6.

This requirement is classified as *core*. However, most of the actions listed that are driven off privileges are

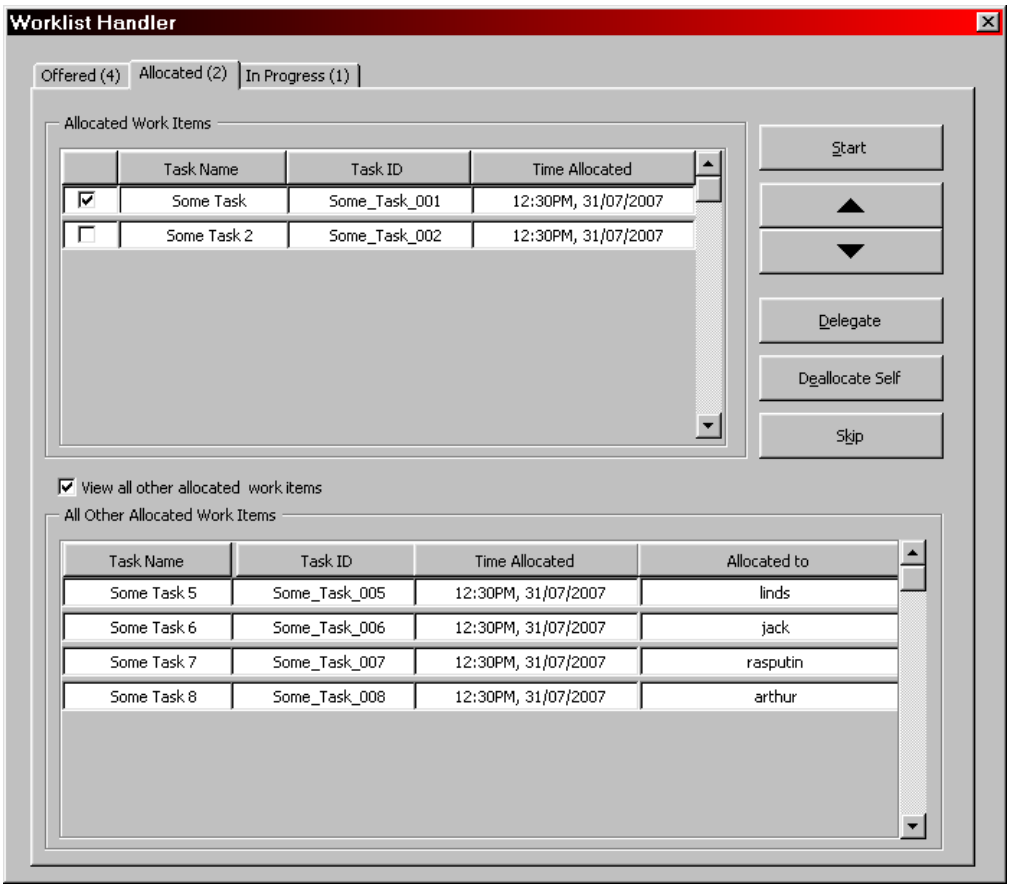


Figure A.6: A possible allocated queue implementation interface

to be considered *non-core*, except **Reorder work item** which is *core*.

REQ-023 : *Workflow Participant to be able to enact stateful reallocation, stateless reallocation, suspension, completion and viewing all started work items from the* **Started** *queue.*

Participants will be able to select a work item from their started queue, and apply one of the following actions (some of which may only be made available when appropriate privileges are granted):

Stateful Reallocation: If the combination of participant and work item has the **Can Reallocate Stateful** privilege enabled, the participant may chose some other participant to reallocate the work item to. The effect of this will be to move a work item, in its current state, to the started queue of the selected participant. The petri-net transition for this is named *process stateful reallocation* in figure 53 of the newYAWL technical report.

Stateless Reallocation: If the combination of participant and work item has the **Can Reallocate Stateless** privilege enabled, the participant may choose some other participant to reallocate the work item to. The effect of this will be to reset a work item to its initial state, and move it to the started queue of the selected participant. The petri-net transition for this is named *process stateless reallocation* in figure 53 of the newYAWL technical report.

Suspend: If the combination of participant and work item has the **Can Suspend** privilege enabled, the participant may choose to suspend the work item. The effect of this will be to move the work item from the started queue to the suspended queue of the participant. The petri-net transition for this is named *suspend* in figure 49 of the newYAWL technical report.

Complete: When a participant chooses to complete a work item, it will be removed from the started queue of the participant, and its state used to complete the work item, progressing the overall workflow as dictated by the specification of the workflow. The petri-net transition for this is named *complete* in figure 50 of the newYAWL technical report.

Toggle view all work items: If the participant has the **Can View All Executing Work Items** privilege set, the participant will be supplied a toggling mechanism, allowing them to view (but not otherwise manipulate) all other started and suspended work items in the entire system. So as not to interfere with their own started and suspended work items, these other work items will appear in a visually distinct queue on the same screen as the participant's own started and suspended work items.

Participants will also be able to open a number of their started work items and work on (edit) their state individually. The actions described above will also be made available to the participant whenever they are working with an individual work item. When the suspend action is chosen, the individual work item screen will become read-only and will only behave as described below.

An indicative screenshot of this requirement is supplied in figure A.7.

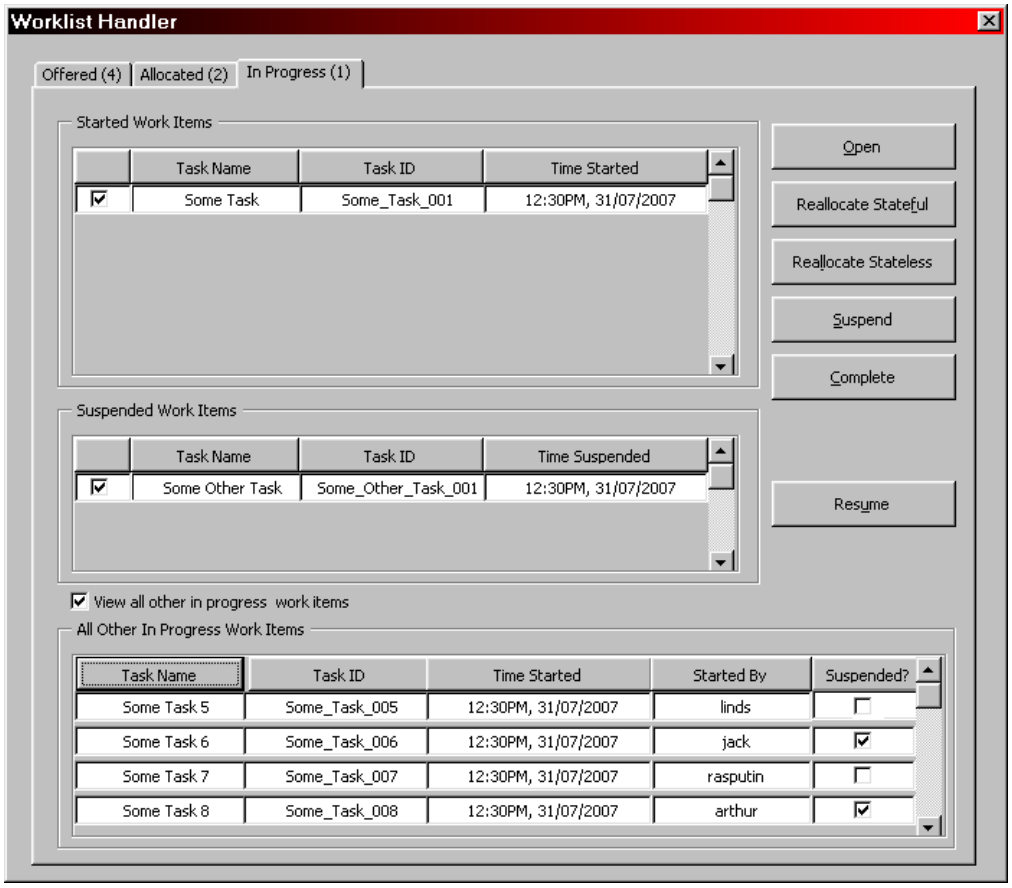


Figure A.7: A possible Started queue implementation interface

This requirement is classified as *core*. The actions listed above except **Suspend** and**Complete** are to be considered *non-core*.

REQ-024 : *Workflow Participant to be able to enact work item resumption from the* **Suspended** *queue.*

Participants will be able to select a a work item from their suspended queue, and apply a **resume** action to that work item. If the participant does not have their **Can Start Concurrent Work Items** privilege enabled,

they may not resume the work item until their started queue is empty. The effect of applying the resume action will be to to move the work item from the suspended queue to the started queue of the participant. The petri-net transition for this is named *resume* in figure 49 of the newYAWL technical report.

Participants will also be able open a number of the suspended work items to view individually in a read-only capacity. The resume action will also be made available to the participant as they view the work item. Choosing the resume action from this individual work item screen will also activate relevant actions that can be applied to work items in the started queue. Of course, this will also have the effect of the participant no longer being able to resume the work item without suspending it again first.

This requirement is classified as *core*.

Automated Workflow Requirements

This section describes the automated requirements of a resource perspective for YAWL that matches the newYAWL technical report.

REQ-025 : *System to support automatically piling work items for a particular task to the relevant participant*

Work items for a particular task can be piled onto a single participant, meaning that all work items spawned for a task across all running cases of a workflow specification of that task will be directly allocated to and automatically started for a single logged-in workflow participant. A number of pre-requisites must exist before the system can pile work items to the participant in this manner:

- Design-time enablement of the **Can Pile** privilege for the participant and task.
- A logged in participant who has indicated that they wish to pile all work items for the selected task to themselves, disallowing any other participant from piling work items of the selected task until they either log off, or indicate that they no longer wish to pile work for the task whilst still being logged in.
- Run-time enablement of the **Can Pile** either through design-time defaulting of enablement for the relevant participant or through explicit enablement by an administrator.

If it is possible for the work item distribution component to pile a work item, it will do so, before it attempts to chain work items, and before it attempts to follow resourcing directives on offering or allocating the work item described at design time. Note that a user logging out will ensure that work will stop being piled to them.

This requirement is classified as *non-core*.

Check models for what happens when the participant logs out as the piling is being evaluated.

REQ-026 : *System to support automatically chaining work items to a relevant participant*

For a participant to chain work items, they must have the privilege **Can Chain Work Item Execution** enabled, and have signaled at logon that they wish to chain the execution of work items. The chaining of work items is the subsequent automatic allocation and starting of work items once the participant finishes some work item. The work item distribution component will need to evaluate the distribution set of possible participants it could have offered new work items to. Should the chaining enabled participant appear in this distribution set, they will be automatically allocated and triggered as having started the new work item. If chaining is not possible for the participant, the work distribution component will default back to offering a new work item to participants in this distribution set.

Note that there is the potential for some subtle unintended side effects in the control flow perspective with a very free applicability of chaining. For example, were we to allow deferred choice through a condition through to two subsequent tasks, one of which does not included the chaining participant, but other does, we have effectively short-circuited workflow, stopping the deferred choice from occurring. Other problems

can be described for multiple-instance tasks starting a chain, the last atomic task(s) within a completing net starting a chain, and even splits off a chain beginning task.

To avoid the above problems, chaining will only be allowed in three basic scenarios:

1. Where one atomic task directly follows another non-branching atomic or multiple-atomic task in a sequence, and the subsequent task is capable of being offered to the chaining participant.
2. Where several atomic tasks follows an atomic or multiple-atomic task with a split, and *all* of the subsequent tasks attached to the split have exactly the same distribution set definition as the split-task.
3. Where several atomic tasks follows a non-split atomic or multiple-atomic task, through a deferred choice condition. *All* outgoing branches of the condition must be to an atomic task that has exactly the same distribution set definition as the split-task.

This requirement is classified as *non-core*.

Check models for what happens when the participant logs out as the chaining is being evaluated.

REQ-027 : *System to support automatically offering work items to relevant participants*

When the resourcing requirements of a task indicates that its work items must be offered via system-initiated offer, the work distribution component must automatically calculate a distribution set of participants as per the workflow specification, and possibly offer a new work item to the final members of this distribution set. As mentioned above, piling and chaining of execution will be attempted first. Should a participant be found via either the piling or chaining mechamism to allocate work to, automated distribution set calculation will not be necessary.

If automated offering is required, as per the specification discussion in section A.2.1, a base distribution set must be built, filtered with the selected filter algorithms, and finally modified to remove participant(s) who last completed a work item of a selected "familiar task" in the same workflow instance if the **Separation of Duties** runtime constraint is active.

In the event that the calculated distribution set is empty (there is nobody to offer the work item to), the work item will be placed on the **Unoffered** queue of workflow administrator(s) for subsequent processing.

If the task has resourcing requirements of user-initiated allocation at this point, the work item will appear on the offered queues of all distribution set participant's worklist handlers. If the task has resourcing requirements of system-initiated allocation, the work item will not appear on any offered queues, but will be subject to the requirement below.

This requirement is classified as *core*.

REQ-028 : *System to support automatically allocating work items to relevant participants*

For those work items whose resource-requirements indicate system-initiated allocation, the work distribution must use the algorithm specified by the workflow designer to automatically choose a single participant from a distribution set of participants, and automatically allocate the work item to that participant.

If the task has resourcing requirements of user-initiated start at this point, the work item will appear on the allocated queue of the chosen participant's worklist handler. If the task has resourcing requirements of system-initiated start, the work item will not appear on their allocated queue, but will be subject to the requirement below.

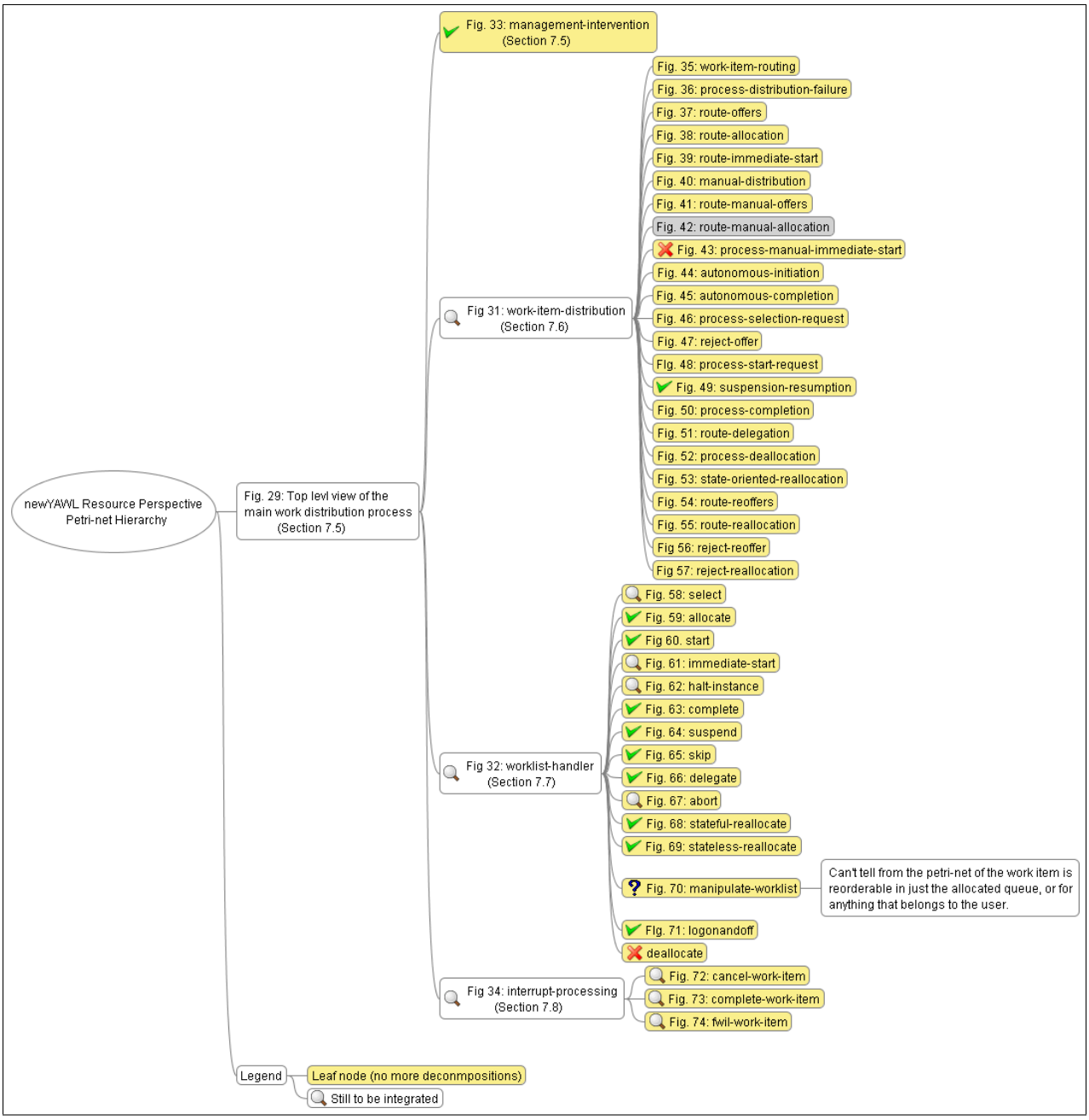
This requirement is classified as *core*.

REQ-029 : *System to support automatically starting work items for relevant participants*

If the task has resourcing requirements of system-initiated start, the moment a work item is allocated to a user, it is also started. It will immediately appear in the started work item queue of that user without having

first gone to their allocated queue.
This requirement is classified as *core*.

Hierarchical View of Resource Perspective Petri-Nets



Bibliography

[1] W.M.P. van der Aalst. The application of Petri nets to workflow management. *Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[2] W.M.P. van der Aalst, B.F. van Dongen, C.W. Günther, R.S. Mans, A.K. Alves de Medeiros, A. Rozinat, V. Rubin, M. Song, H.M.W. Verbeek, and A.J.M.M. Weijters. ProM 4.0: Comprehensive Support for Real Process Analysis. In J. Kleijn and A. Yakovlev, editors, *Application and Theory of Petri Nets and Other Models of Concurrency (ICATPN 2007)*, volume 4546 of *Lecture Notes in Computer Science*, pages 484–494. Springer, 2007.

[3] W.M.P van der Aalst and K.M. van Hee. *Workflow Management: Models, Methods and Systems*. MIT Press, Cambridge, MA, USA, 2002.

[4] W.M.P. van der Aalst and A.H.M. ter Hofstede. Workflow Patterns: On the Expressive Power of (Petri-net-based) Workflow Languages. In Kurt Jensen, editor, *Proceedings of the Fourth International Workshop on Practical Use of Coloured Petri Nets and the CPN Tools*, volume 560 of DAIMI, pages 1–20, Aarhus, Denmark, August 2002. University of Aarhus.

[5] W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet another workflow language. *Information Systems*, 30(4):245–275, 2005.

[6] Michael Adams. *Facilitating Dynamic Flexibility and Exception Handling for Workflows*. PhD Thesis, Queensland University of Technology, Brisbane, Australia, 2007. Available through http://www.yawl-system.com.

[7] Michael Adams, Arthur H.M. ter Hofstede, Wil M.P. van der Aalst, and David Edmond. Dynamic, Extensible and Context-Aware Exception Handling for Workflows. In Robert Meersman and Zahir Tari, editors, *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS, OTM Confederated International Conferences CoopIS, DOA, ODBASE, GADA, and IS 2007, Vilamoura, Portugal, November 25-30, 2007, Proceedings, Part I*, volume 4803 of *Lecture Notes in Computer Science*, pages 95–112. Springer, 2007.

[8] Michael Adams, Arthur H.M. ter Hofstede, David Edmond, and W.M.P. van der Aalst. Worklets: A service-oriented implementation of dynamic flexibility in workflows. In R. Meersman and Z. Tari et. al., editors, *Proceedings of the 14th International Conference on Cooperative Information Systems (CoopIS'06)*, volume 4275 of *Lecture Notes in Computer Science*, pages 291–308, Montpellier, France, November 2006. Springer-Verlag.

[9] M. de Leoni, W.M.P. van der Aalst, and A.H.M. ter Hofstede. Visual Support for Work Assignment in Process-Aware Information Systems. In M. Dumas, M. Reichert, and M.-C. Shan, editors, *BPM 2008*, volume 5240 of *Lecture Notes in Computer Science*. Springer, 2008.

[10] A.H.M. ter Hofstede, W.M.P van der Aalst, M. Adams, and N. Russell, editors. *Modern Business Process Automation: YAWL and its Support Environment*. Springer, 2010.

[11] T. Murata. Petri nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[12] J.L. Peterson. *Petri net theory and the modeling of systems*. Prentice-Hall, Englewood Cliffs, USA, 1981.

[13] A. Rozinat, M. Wynn, W.M.P. van der Aalst, A.H.M. ter Hofstede, and C. Fidge. Workflow Simulation for Operational Decision Support Using Design, Historic and State Information. In M. Dumas, M. Reichert, and M.-C. Shan, editors, *BPM 2008*, volume 5240 of *Lecture Notes in Computer Science*, pages 196–211. Springer, 2008.

[14] N. Russell, W.M.P van der Aalst, and A.H.M. ter Hofstede. Workflow exception patterns. In E. Dubois and K. Pohl, editors, *Proceedings of the 18th International Conference on Advanced Information Systems Engineering (CAiSE'06)*, volume 4001 of *Lecture Notes in Computer Science*, pages 288–302, Luxembourg, Luxembourg, 2006. Springer.

[15] N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P van der Aalst. newYAWL: achieving comprehensive patterns support in workflow for the control-flow, data and resource perspectives. Technical Report BPM-07-05, BPM Center, 2007. http://www.BPMcenter.org.

[16] N.C. Russell. *Foundations of Process-Aware Information Systems*. PhD Thesis, Queensland University of Technology, Brisbane, Australia, 2007. Available through http://www.yawl-system.com.

[17] H.M.W. Verbeek, Wil M.P. van der Aalst, and Arthur H.M. ter Hofstede. Verifying Workflows with Cancellation Regions and OR-joins: An Approach Based on Relaxed Soundness and Invariants. *Computer Journal*, 50(3):294–314, 2007.

[18] M. Weske. *Business Process Management: Concepts, Languages, Architectures* . Springer, 2007.

[19] Moe Thandar Wynn. *Semantics, Verification, and Implementation of Workflows with Cancellation Regions and OR-joins*. PhD Thesis, Queensland University of Technology, Brisbane, Australia, 2006. Available through http://www.yawl-system.com.

[20] M.T. Wynn, D. Edmond, W.M.P. van der Aalst, and A.H.M. ter Hofstede. Achieving a general, formal and decidable approach to the OR-join in workflow using Reset nets. In G. Ciardo and P. Darondeau, editors, *Proceedings of the 26th International Conference on Application and Theory of Petri nets and Other Models of Concurrency (Petri Nets 2005)*, volume 3536 of *Lecture Notes in Computer Science*, pages 423–443, Miami, USA, 2005. Springer-Verlag.