Defining Workflows

Version 2018.1
2019-09-20

InterSystems Corporation 1 Memorial Drive Cambridge MA 02142 www.intersystems.com

Defining Workflows
Ensemble Version 2018.1 2019-09-20

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)
Tel: +1-617-621-0700
Tel: +44 (0) 844 854 2917
Email: support@InterSystems.com

Table of Contents

About This Book

This book explains the operation of the Ensemble Workflow Engine and describes how to use it to build workflow into an Ensemble integration solution.

This book contains the following sections:

- Overview of Workflow in Ensemble
- Developing a Workflow
- Including Custom Features in a Workflow
- Testing a Workflow
- Exploring the Workflow Sample
- Available Workflow Metrics

For a detailed outline, see the table of contents.

The following books provide related information:

- *Ensemble Best Practices* describes best practices for organizing and developing Ensemble productions.
- *Developing Ensemble Productions* explains how to perform the development tasks related to creating an Ensemble production.
- *Configuring Ensemble Productions* explains how to perform the configuration tasks related to creating an Ensemble production.
- *Managing Ensemble* describes how to use the Ensemble pages of the Management Portal. In particular, see the chapter "Managing Workflow Roles, Users, and Tasks."

For general information, see the *InterSystems Documentation Guide*.

1

Overview of Workflow in Ensemble

This chapter provides an overview of workflow features in Ensemble. It discusses the following topics:

1.1 Introduction

A *workflow management system* automates the distribution of tasks among users. Automating the distribution of tasks according to a predefined strategy makes task assignment more efficient and task execution more accountable. A typical example is a help desk application that accepts problem reports from customers, routes the reports to members of the appropriate organizations for action, and upon resolution of the problem, reports the results to the customer.

The Ensemble Workflow Engine provides a much higher level of functionality than traditional, stand-alone workflow management systems, as the next subsections explain.

1.1.1 Integration into Ensemble

The Ensemble Workflow Engine takes full advantage of the Ensemble architecture. As a result, it can seamlessly interoperate with enterprise applications, technology, and data, as well as human participants. Highlights of this collaboration include the following:

- "Straight-through" business processes (that is, completely automated business processes) can incorporate human involvement to handle exception cases, such as approval for especially large orders.
- An Ensemble workflow can call out to enterprise applications: to notify them of events within the human workflow, or to obtain additional information needed by the human workflow.
- Persistence: The Workflow Engine leverages Ensemble persistent storage features to support long-running business processes that may take days or weeks to complete.

1.1.2 Support for Composite Applications

Ensemble provides a richly stocked development environment that allows developers to easily create composite applications that span disparate systems and technologies across the enterprise. Because the Workflow Engine is fully integrated within Ensemble:

- Composite applications can easily incorporate complex manual interactions that reach across geographical, technological, and departmental divisions.
- User-based process definitions can be separated from business logic, allowing developers and analysts to define each segment distinctly within a cohesive whole.
- Workflow systems are more versatile, more powerful, easier to create, and simpler to maintain.

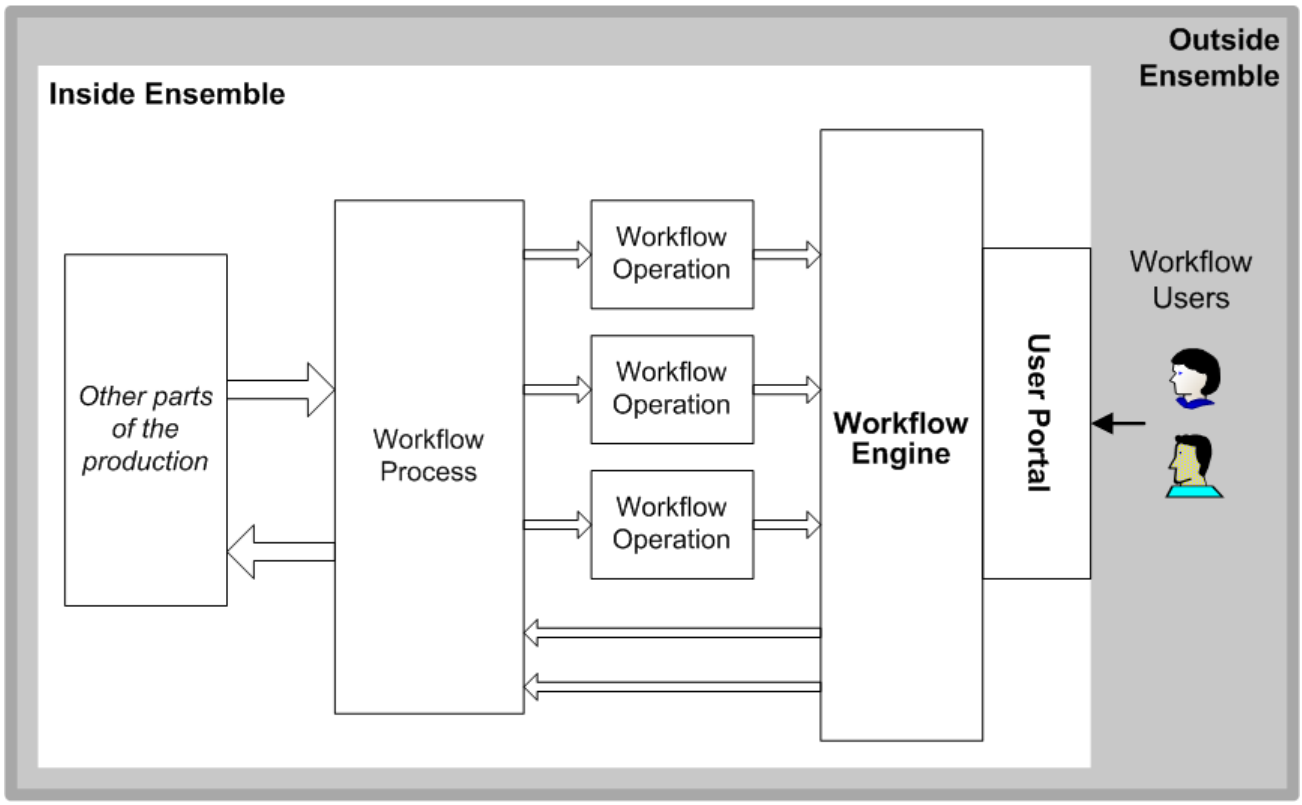
1.1.3 Productivity Features

The Ensemble Workflow Engine provides automatic integration with productivity tools such as the following:

- Business Process Designer: Workflow analysts can create workflows using the Business Process Designer, the Ensemble graphical editing tool that automatically generates full-fledged, working code from a business process diagram.
- Business Activity Monitoring: Developers can easily create corporate dashboards and event triggers to display current workflow status to enterprise analysts.
- Visual Trace: All tasks are sent as messages, so system administrators can view the status of tasks using the powerful Visual Trace message tracing tool.

1.2 Workflow Components in a Production

The following figure shows workflow components in a production:



A *workflow process* is a BPL process that sends task requests to workflow operations. This process is responsible for coordinating work among these workflow operations. It might also invoke standard business operations (that is, ones that do not require human intervention). Both types of element may be required to complete the business process.

A *workflow operation* is a special-purpose business operation that represents a workflow role (discussed in the next heading). The workflow operation sends tasks, via the Workflow Inbox, to all users who belong to the associated role.

A workflow process sends a task request to a workflow operation in exactly the same way as business processes send other kinds of request messages to other kinds of business operation.

In Ensemble terms, a *workflow task* is an item of work that is performed offline in support of an ongoing business process. Ensemble *represents* a task with a special-purpose Ensemble message object called a *task request* that carries information about that work. The workflow process sends task requests and receives task responses. As with all other messages, these messages are saved in the database until purged (if ever).

1.3 Workflow Roles and Users

In a typical organization, multiple people might be able to fulfill a certain kind of task. Therefore, to organize distribution of tasks to people, Ensemble uses *workflow roles*. A workflow role is a list of users that can fulfill a certain kind of task.

In theory, there may be any number of workflow roles defined within an enterprise. In practice, roles map to departments or job positions within in the enterprise, such as Accounting, Finance, Teller, Manager, Supervisor, or Director.

Any number of *workflow users* may be members of a workflow role. Assignment of users to roles should make sense from a practical and organizational standpoint. If a person is authorized to perform a task, that person should be a member of the corresponding workflow role.

Workflow roles are defined per Ensemble namespace, not per production. This means that the same workflow role definitions are available to all the productions in the same Ensemble namespace. The same is true for workflow users when they correspond to Ensemble user accounts (as is typically done).

1.4 User Interfaces for Workflow

Ensemble now provides two user interfaces to support workflow. These are intended for different sets of users.

1.4.1 Implementers and Supervisors

The Management Portal provides pages that implementers and supervisors can use to manage workflow roles, users, and tasks. To access them, select **Ensemble**, click **Manage**, and then click **Workflow**.

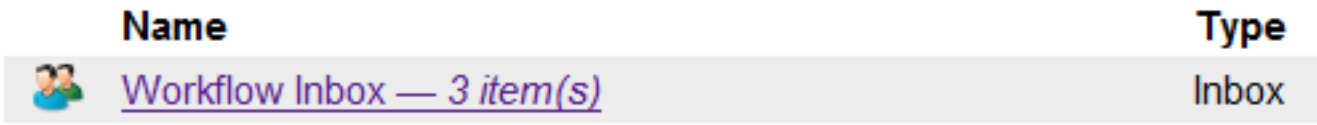
These pages are not meant for end users. Supervisors can assign or cancel tasks, but other actions (such as marking tasks complete) are not available here.

For details, see "Managing Workflow Roles, Users, and Tasks" in *Managing Ensemble*.

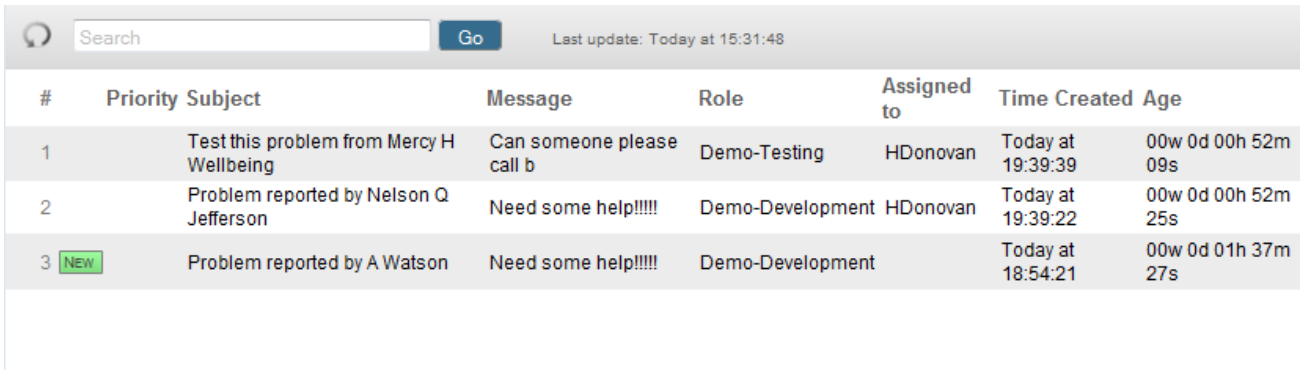
1.4.2 End Users

Users manage their workflow tasks within the DeepSee User Portal, which also displays Ensemble dashboards (and DeepSee dashboards). The DeepSee User Portal is accessible from the Management Portal, but it is more likely that your application would provide a link directly to it. (Also note that the User Portal does not have a title; it is suitably generic for all users.)

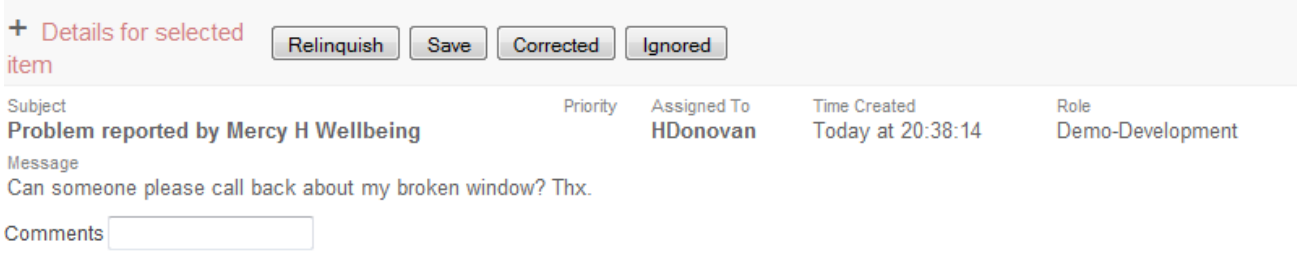
For an Ensemble workflow user, the main area of the User Portal displays the Workflow Inbox. For example:



When the user clicks **Workflow Inbox**, the User Portal displays something like the following:



When a user clicks a task, the system displays the corresponding *task form*, which you can configure or customize. For example:



For details, see "Using the Portal Features" in the *DeepSee End User Guide*.

1.5 Life Cycle of a Task

The life cycle of a task within an Ensemble production is as follows:

1. A workflow process receives some input and then creates a task request (a message).
2. The workflow process sends the task request to a workflow operation.
3. The workflow operation submits the task request to the Workflow Engine.
4. The Workflow Engine instantiates the appropriate task *response* object. This object includes the fields from the original task request, and oversees task distribution to some user.
5. Task distribution follows either the default strategy or a custom distribution strategy. See "Task Distribution Options," later in this chapter, for details.
6. Once the task is assigned and completed, the Workflow Engine returns the task response to the business process.

1.6 Task Forms and Customization Options

The DeepSee User Portal (mentioned earlier) provides a Workflow Inbox for each user. When a user clicks a task, the system displays a form that is specific to that task. You can customize this form, as follows:

- The form is generated from metadata, which may be contained within reusable template files or defined at runtime by enterprise staff.
- The form is easily styled to fit within corporate standards.
- If the enterprise wishes to use in-house technologies to drive the user experience, Ensemble workflow tasks can be exposed to such technologies in a variety of ways, via XML documents, Web services, .NET, Java, C++ objects, or as relational structures.

1.7 Task Distribution Options

When a workflow operation receives a task request, it turns the request over to the Ensemble Workflow Engine to distribute the task to one of the users defined for that role. The strategy used to distribute tasks can be the Ensemble default, which works as follows:

- For each workflow user, there is a *worklist* that lists all the tasks currently assigned to or associated with that user. Only active tasks for the specific user appear on the worklist. Once the task is completed, discarded, cancelled, or reassigned to another user, it disappears from the list.
- The Workflow Engine posts any task requests that it receives on the worklist for each user in the workflow role to which the request was sent. When a task is first posted, it has the status Unassigned. The task is said to be *associated* with all these users. While it remains Unassigned, each of the associated users sees the task on his or her worklist.
- It is possible for a user to acquire ownership of a task through assignment by a supervisor, or by default, but in general the user must accept the task to acquire ownership. Any user in the workflow role can accept any task on his or her worklist, on a first-come, first-served basis, using the Workflow Inbox.
- Once a user has acquired ownership of a task, the task is said to be *assigned* to that user. The status of the task changes from Unassigned to Assigned. The task remains in the Workflow Inbox for the user who accepted it, but disappears from the Workflow Inboxes of the other users in that workflow role.
- A task request has a property that allows it to express a preference for a specific user. If that user is inactive or not defined at the time the task request is received, the Workflow Engine returns an error to the workflow operation and the task is not performed.
- A supervisor can reassign an active task from the workflow pages of Management Portal. In this case, the task leaves the worklist for the previously assigned user and appears on the worklist for the newly assigned user.
- The assigned user may relinquish a task without completing it. If a user relinquishes a task unfinished, the task reverts to its original Unassigned status, and the Workflow Engine posts it to the Workflow Inbox for each associated user (each user in the workflow role).
- If a supervisor deletes a user definition or marks a user inactive while workflow is proceeding, the result is the same as if that user suddenly relinquished all of his or her tasks. That is, each task that was on the user's worklist acquires an Unassigned status, and the Workflow Engine posts each task on the Workflow Inbox of every user in the workflow role to which the task was originally sent.
- If a supervisor deletes a workflow role definition while workflow is proceeding, outstanding requests to that role are allowed to complete, but future requests to that role cause an error, due to a mismatch between the workflow operation name and the (nonexistent) role definition name.
- Each task request specifies one or more *actions*. Each action is a string that the associated task form displays as a button: Accept, Relinquish, and so on. The assigned user can click one of these buttons to complete the task and send a response back to the business process.
- When the assigned user clicks an action button for a task, the Workflow Engine fills in the various properties of the task response object — including the name of the user-selected action — and returns this response to the workflow operation, which relays it to the original requesting business process. The Workflow Engine marks the task Completed and removes it from all worklists.
- After completion, a full record of any task remains in the message warehouse for the production, and are visible from the Management Portal. Task details relating directly to workflow are visible from the Workflow Management Portal. This is true even if the message was not properly completed, but was discarded or cancelled, or encountered an error.

The strategy used to distribute tasks is embedded in the **OnNewTask()** method of the task response class EnsLib.Workflow.TaskResponse. You can subclass this class and the methods to change the logic, if desired. For instructions, see the chapter "Including Custom Features in a Workflow."

1.7.1 Custom Task Distribution

Workflow designers can make use of a wide variety of task distribution strategies within a workflow definition. The following strategies are built into the Ensemble Workflow Engine:

- First come, first served (FCFS)
- Assignment by job title
- Assignment by name
- Assignment by a user-defined ranking
- Assignment by current user workload
- ...and more.

It is also possible to create new, custom task distribution strategies.

2

Developing a Workflow

This chapter describes how to develop a workflow and add it to a production. The next chapter provides details on optional customizations you can perform during development.

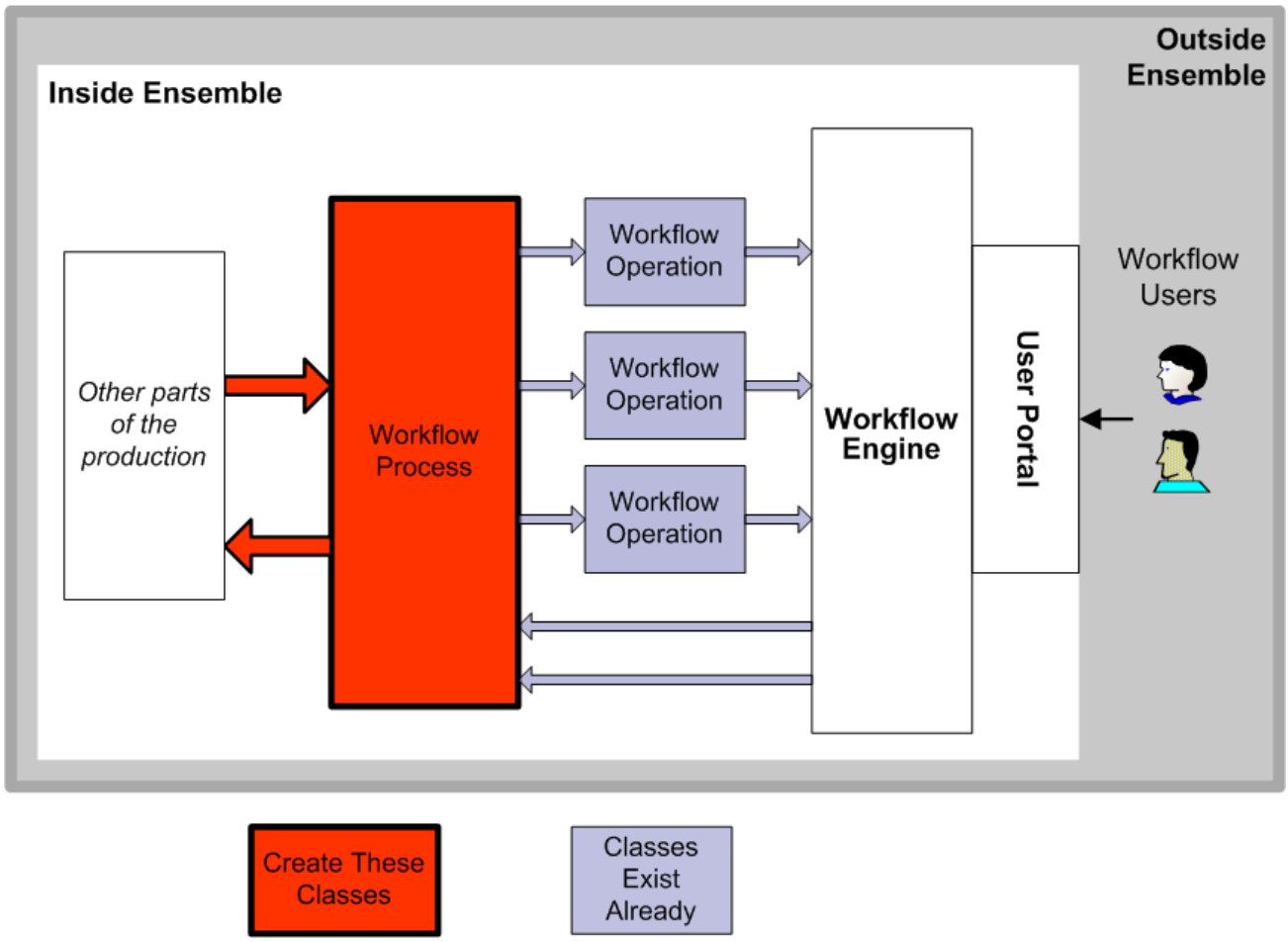
This chapter includes the following topics:

- Overview
- Designing the Workflow Process
- Creating and Adding the Workflow Process
- Adding the Workflow Operations to a Production
- Next Steps

The appendix "Exploring the Workflow Sample" presents a simple example, available within the ENSDEMO namespace.

2.1 Overview

The following diagram shows workflow elements in a production:



These components are as follows, from left to right:

- A message request class carries the specific details of the workflow from its originator elsewhere in the production. This is a standard message request class that you define, that is, a subclass of Ens.Request or some (less often) another persistent class.
- A message response class carries the specific details of the workflow back to the originator, once the task has been completed or otherwise handled. This is a standard message response class that you define, that is, a subclass of Ens.Response or some (less often) another persistent class.

  This book does not provide any special details on defining this class or the corresponding request class, because the implementation is completely dependent on your needs.
- A *workflow process* that serves as the coordinator of the communications. To create this:

  1. Use the Business Process Designer and create a business process based on the class Ens.BusinessProcess.BPL. It must contain the logic to receive the inbound messages and to call (asychronously) to business operations in the same production, as needed.
  2. Add the business process to the production as usual.
- A special message class EnsLib.Workflow.TaskRequest carries requests from the business process.

  Likewise, the special message class EnsLib.Workflow.TaskResponse carries replies from the Workflow Engine.

  You can create and use subclasses, but that is not generally necessary, because these classes are designed to carry information in a flexible manner.

- Each *workflow operation* receives task requests and communicates with the Workflow Engine, which makes tasks visible to specific users. In particular, each workflow operation has exactly one associated workflow role. The workflow operation makes tasks visible to all users who belong to that workflow role; other users do not see these tasks.

  To create each of these, you add a business operation based on EnsLib.Workflow.Operation. Little configuration is needed.

  Or you can subclass EnsLib.Workflow.Operation and implement its callback methods to interact with the Workflow Engine in a custom way.

2.2 Designing the Business Process

The first step is to design the business process class and make some key decisions.

1. Identify where calls to workflow (human processes) will occur.
2. Each of these calls is a task; *name each task*.
3. *Name each workflow role* to which a task is sent.
4. Decide on a task distribution strategy for each task (perhaps the default strategy).
5. Create any message types needed to invoke the business process.

   This book does not provide any special details on defining these messages, because the implementation is completely dependent on your needs.
6. Create the business process class as described in the next section.

2.3 Creating the Workflow Process

To create the workflow process, use the Business Process Designer as described in *Developing BPL Processes*.

For each task in the business process:

- Provide a <call> to invoke the workflow operation asynchronously.
- In the <call> identify the destination workflow operation by name. Each workflow operation is associated with exactly one workflow role, which in turn corresponds to a configurable set of users.
- In the <call> use the correct task request object (EnsLib.Workflow.TaskRequest or a subclass) and sets its properties.

  The <call> should also examine the task response object (EnsLib.Workflow.TaskResponse or a subclass), and use it to set the BPL *context* variable for use in later processing.
- Provide a <sync> to catch the results of the asynchronous <call>.

For an example, see the appendix "Exploring the Workflow Sample."

2.3.1 Defining the Task Request

Each <call> in the process should use EnsLib.Workflow.TaskRequest or a subclass as its message class. To define the request, the <call> should set properties of the *callrequest* variable.

The following table lists the available properties in EnsLib.Workflow.TaskRequest. These properties are all optional, but of course your business process must send all the information that the Workflow Engine needs.

Property	Purpose
%Actions	Optional. Comma-delimited list of Actions defined for this task. For example: "Approve,Reject,NeedMoreInfo" These determine the action buttons displayed when a user reviews a task. The selected user action is returned as part of the task response.
%Command	Optional. Command string to be passed to the Workflow Engine. You can use this to provide additional data-driven custom behavior for tasks.
%FormTemplate	Optional. Name of a Caché Server Pages (CSP) page that provides the HTML form template for this task.
%FormFields	Optional. Comma-separated list of fields that should appear in the form associated with this task.
%FormValues	Optional. Collection of values to display within the form displayed for this task.
%Message	Optional. Detailed message body for this task. This is displayed when the user reviews a task.
%Priority	Optional. Priority of the requested task: 1 is highest. This value is used to sort items in the user worklist.
%Subject	Optional. Short summary of this task. This is displayed in the User worklist.
%TaskHandler	Optional. Name of response class that is used to manage the distribution of this task. It is also used as the response type for this request. The class named in %TaskHandler must be a subclass of EnsLib.Workflow.TaskResponse.
%Title	Optional. The name of the title within the given that is preferred for handling this task. Whether or not this user actually is assigned to the task depends on how the distribution strategy used for this task.
%UserName	Optional. The name of the user that is preferred for handling this task. Whether or not this user actually is assigned to the task depends on how the distribution strategy used for this task.

2.3.2 Using the Task Response

As noted earlier, the <call> for a given task should also examine the task response object (EnsLib.Workflow.TaskResponse or a subclass), and use it to set the BPL *context* variable for use in later processing. EnsLib.Workflow.TaskResponse defines the following properties:

Property	Purpose
%Action	Optional. Once the task is complete, this property will contain the value of the action that the user selected to complete the task. See %Actions.
%Actions	Optional. A comma-delimited list of actions defined for this task. For example: "Approve,Reject,NeedMoreInfo" These determine the action buttons displayed when a user reviews a task. The selected user action is returned in the %Action property of the task response.
%FormFields	Optional. Comma-separated list of fields that should appear in the form associated with this task. This is a copy of the value provided from the initial task request.

Property	Purpose
%FormTemplate	Optional. Name of the CSP page that provides the form template for this task. This is a copy of the value provided from the initial task request.
%FormValues	Optional. A collection of values from the form associated with this task (if any).
%Message	Optional. Detailed message body for this task. This is a copy of the value provided from the initial task request.
%Priority	Optional. The priority of the requested task: 1 is highest. This is a copy of the value provided from the initial task request.
%RoleName	Optional. The name of the role that handled this task. This is a copy of the value provided from the initial task request.
%Status	Optional. The external status of this task. Used to query the current status of a task.
%Subject	Optional. Short summary of this task. This is a copy of the value provided from the initial task request.
%TaskStatus	Optional. The internal status of this task (Assigned, Unassigned, Cancelled, Discarded, Deleted). Used by the Workflow Engine to manage the task. Custom code should not modify the value of this property.
%UserName	Optional. The name of the user that last handled this task. This value is set when the task is completed.
%UserRanking	Optional. The ranking associated with the user that last handled this task (if the user has a role-assigned ranking).
%UserTitle	Optional. The title associated with the user that last handled this task (if the user has a role-assigned title).

2.3.3 Adding the Workflow Process to the Production

To add the workflow process to a production, do the following:

1. Display the production in the **[Ensemble] > [Production Configuration]** page of the Management Portal. To access this page, click **Ensemble**, click **Configure**, click **Production**, and then click **Go**.
2. In the **Processes** column, click the Add button (a plus sign).

   Ensemble displays a dialog box.
3. For **Business Process Class**, select the class you created in the Business Process Editor.
4. For **Process Name**, type a suitable name.
5. Click **OK**.

For information on other settings, see *Configuring Ensemble Productions*.

2.4 Adding the Workflow Operations to a Production

To add a workflow operation to a production, do the following:

1. Display the production in the **[Ensemble] > [Production Configuration]** page of the Management Portal. To access this page, click **Ensemble**, click **Configure**, click **Production**, and then click **Go**.
2. In the **Operations** column, click the Add button (a plus sign).

   Ensemble displays a dialog box.
3. Click the **Workflow** tab.
4. Select the EnsLib.Workflow.Operation class from the **Class Name** list.

   Or select your custom subclass.
5. For **Operation Name**, type the name of this business operation. The name must match the target attribute in the corresponding <call> from the BPL business process.
6. Optionally click **Auto-Create Role**. If enabled, this option causes Ensemble to automatically create a workflow role whose name is identical to **Operation Name**.

   You can enable this setting later, if wanted.
7. Click **OK**.

For information on other settings, see *Configuring Ensemble Productions*.

2.5 Next Steps

- Associate workflow users with the appropriate workflow roles. See "Managing Workflow Roles, Users, and Tasks" in *Managing Ensemble*.
- Test the workflow as described in the chapter "Testing a Workflow."
- Optionally add a dashboard so supervisors can monitor workflow progress. Ensemble automatically collects certain workflow metrics. The general development and configuration tasks are as usual:

  1. Create one or more business metric classes. See "Defining Business Metrics" in Developing Ensemble Productions.

     Each class can invoke methods of the class EnsLib.Workflow.Engine, which provides methods that report statistics about workflow roles.

     For quick reference, these are listed in the appendix "Available Workflow Metrics."
  2. When configuring the production, add each business metric as a business service.
  3. Define dashboards. See "Defining Dashboards" in Configuring Ensemble Productions.

3

Including Custom Features in a Workflow

This chapter describes how to include custom features in a workflow. It includes the following topics:

3.1 Extending the Standard Task Form

Each task has an associated task form that displays (at a minimum) a standard set of items, as follows:

- The read-only **TaskId**, **Owner**, %Subject and %Message properties of the task
- The **Accept** button (when the task is unassigned)
- The **Relinquish** and **Save** buttons (when the task is assigned)

To add additional buttons (which are visible only when the task is assigned), provide a comma-separated list of button names in the %Actions property of the task request.

To add additional editable values (which are visible only when the task is assigned), set one of the following properties in the task request object:

- %FormFields — Provides a list of fields for the task form to display.

  The task request and response objects have a %FormFields property. Upon making the call to a task request, set the %FormFields value to a comma-delimited list of the fields that you want to display on the task form. For now, these are all assumed to be simple string values (if you need more control, use a form template).

- %FormTemplate — Identifies a Caché Server Pages (CSP) file that defines the HTML form template to be displayed.

  The task request and response objects have a %FormTemplate property. Upon making the call to a task request, set the %FormTemplate value to the name of a CSP file that defines a form to be displayed. This CSP page should only define the form and contents: it should *not* have HTML, HEAD, or BODY sections. The CSP page can overload certain JavaScript Event methods. When this form is processed, the variable *%task* is the current task response object.

If you use %FormFields or %FormTemplate, you can specify default values to be shown in the fields. To do so, specify the %FormValues property of the task request object. This property is an array of strings, subscripted by field name. Use the array interface to add values to the property.

When the Workflow Engine processes a task, it makes a copy of all the form-related properties from the task request into the task response, including the array collection %FormValues. As fields in the form acquire new values, the Workflow Engine keeps updating the values in the %FormValues collection of the task response object.

Forms always display the current values in the %FormValues collection. This allows dynamic operation of forms within a specific task: by modifying the values of the form-related properties, the task response callback methods can influence how form processing takes place. The %FormValues collection of the task response object is available to the original caller when the final task response is sent back to it.

3.2 Using a Custom Task Form

It is possible to display a custom form to a user for a workflow task. The steps are as follows:

1. Create an HTML template file that defines the contents of the customized form. This is implemented as a Caché Server Pages (CSP) page so that it can contain dynamic content, if desired.
2. Set the %FormTemplate property of the task request object to the name of this CSP page. For example, the <call> statement sending the task request to the workflow operation would contain:

   ```
   <assign property='callrequest.%FormTemplate'
           value='"MyForm.csp"'
           action='set' />
   ```

3. Set the %FormFields property of the task request object to a comma-delimited list of field names. This is the list of fields that will be defined for the form. For example:

   ```
   <assign property='callrequest.%FormFields'
           value='"Details,CustomerName"'
           action='set' />
   ```

4. If you wish to provide initial values for the form fields, set the corresponding element (that is,. the array key is the name of the field) of the %FormValues collection property of the task request object to the desired value. For example:

   ```
   <assign property='callrequest.%FormValues'
           value='request.CustomerName'
           action='set'
           key='"CustomerName"' />
   ```

As described in previous sections, whenever a workflow user reviews a task that he or she currently owns, the system displays a the associated task form. By default, this form is automatically generated using the fields defined by the values in the %FormFields property of the task request object. The calling business process provides these values. However, there is a way to use a custom-designed template instead of this generated form.

A workflow form template defines a block of HTML that is injected into the standard task form. This HTML block can include any number of form fields; these fields will be automatically submitted and processed by the Workflow Engine whenever a user performs an action on the workflow task. The HTML contained in a template file is not a complete HTML document; it is simply the HTML needed to display the custom portion of the form. Specifically, the resulting HTML generated by the task form looks like this:

```
<html>
  <body>
    <form>
      —TEMPLATE CONTENTS INJECTED HERE—
    </form>
  </body>
</html>
```

The following template defines an HTML table that displays two HTML input controls: a text box and a select (combobox). The names of the controls correspond to the fields defined by the %FormFields property. This example also uses a server-

side expression to get the initial value for the Details property. The *%task* variable is always pre-set to the current task response object:

```
<!– workflow template –>
<table>
  <tr>
    <td>Details:</td>
    <td>
      <input type="text"
             name="Details"
             value="#(%task.%FormValues.GetAt("Details"))#">
    </td>
  </tr>
  <tr>
    <td>Company:</td>
    <td>
      <select name="CustomerName">
        <option value="ABC Corp">ABC Corp</option>
        <option value="XYZ Corp">XYZ Corp</option>
      </select>
    </td>
  </tr>
</table>
```

In addition to HTML, the workflow template file can contain the following, optional, Javascript callback functions:

- **onLoad()** — called when the workflow form is loaded into the browser.
- **onAction()** — called when the user clicks on one of the action buttons displayed by the form.

To add these callbacks to a workflow template file, simply place the function definition within a <script> tag in the template file; for example:

```
<script language="JavaScript">
        function onLoad(form)
        {
                // form is the workflow form object
                return true;
        }

        function onAction(form,action)
        {
                // form is the workflow form object
                // action is a string containing the user's action
                // returning false will cancel this action
                return true;
        }
        </script>
```

3.3 Customizing the Task Distribution Strategy

The task distribution strategy is specified in the task response class. To implement a custom task distribution strategy, do the following:

- Create a subclass of EnsLib.Workflow.TaskResponse and override its **OnNewTask()** callback method (and possibly other methods). The first subsection provides details on the options.
- Make sure to invoke the custom task response class. This is discussed in the second subsection.

3.3.1 Creating a Custom Task Response Class

When you create a subclass of EnsLib.Workflow.TaskResponse, you override its **OnNewTask()** callback method (and possibly other methods). The following is a list of the callback methods that you can override:

- **OnAction()** — Called when a user selects an action from the worklist form. Typically this marks the end of a task.

- **OnAssign()** — Called when a user requests ownership of a task associated with it. Typically this method performs the assignment.
- **OnCancel()** — Called when a task is cancelled, for example when it times out.
- **OnFormSubmit()** — Called when the task form associated with this task is submitted.
- **OnNewTask()** — Called when a new task is received by the Workflow Engine. Typically this method associates the task with members of the current role.
- **OnRelinquish()** — Called when a user requests giving up ownership of a task associated with it. Typically this method unassigns the task and then sends the task back to others in the role.
- **OnRoleChange()** — Called when a user or role definition associated with this task is changed, for example when the list of users within a role changes.

EnsLib.Workflow.TaskResponse callback methods control the distribution of tasks by calling a number of API methods defined in the Workflow Engine (EnsLib.Workflow.Engine).

The following is a list of the class methods in EnsLib.Workflow.Engine that you can use when you override the callbacks listed previously.

1. **AssignTask()** — Assign a task to a specific user.
2. **CompleteTask()** — Mark a task as Completed and return the response to the caller.
3. **FindLeastBusyUser()** — Return the name of the “least busy” user. This is the user with the fewest assigned tasks in the system.
4. **SendTask()** — Send a task to a specific user.
5. **SendTaskToAll()** — Send a task to all users in the current role.
6. **SendTaskToTitle()** — Send a task to one or more users with the given title in the current role.
7. **SendTaskToTop()** — Send a task to the top *n* users in the current role, according to their respective ranking in the role.
8. **UnassignTask()** — Remove task assignment.

For details on these methods, see the *InterSystems Class Reference*.

3.3.2 Invoking the Custom Task Response Class

A task request message has class parameter (*RESPONSECLASSNAME*) that specifies which response class to use. You can override this by setting the %TaskHandler property of the request class.

This means that you have two ways to cause the task request to use the desired task response:

- Optionally create a subclass of EnsLib.Workflow.TaskRequest and override its *RESPONSECLASSNAME* parameter to equal the name of your custom task response class. For example:

  ```
  Class MyApp.MyWorkflowRequest Extends EnsLib.Workflow.TaskRequest
  {
  Parameter RESPONSECLASSNAME = "MyApp.MyWorkflowResponse";
  }
  ```

  Then use this message class in the appropriate parts of the workflow process.

- In the workflow process, when making the call to the workflow operation, set the value of the %TaskHandler property of the request instance so that it gives the name of the desired task response class name. For example:

  ```
  set callrequest.%TaskHandler="MyApp.MyWorkflowResponse"
  ```

4

Testing a Workflow

This chapter discusses items relevant to testing your workflows.

4.1 Test Checklist

At a minimum, when you test the workflow, test these items:

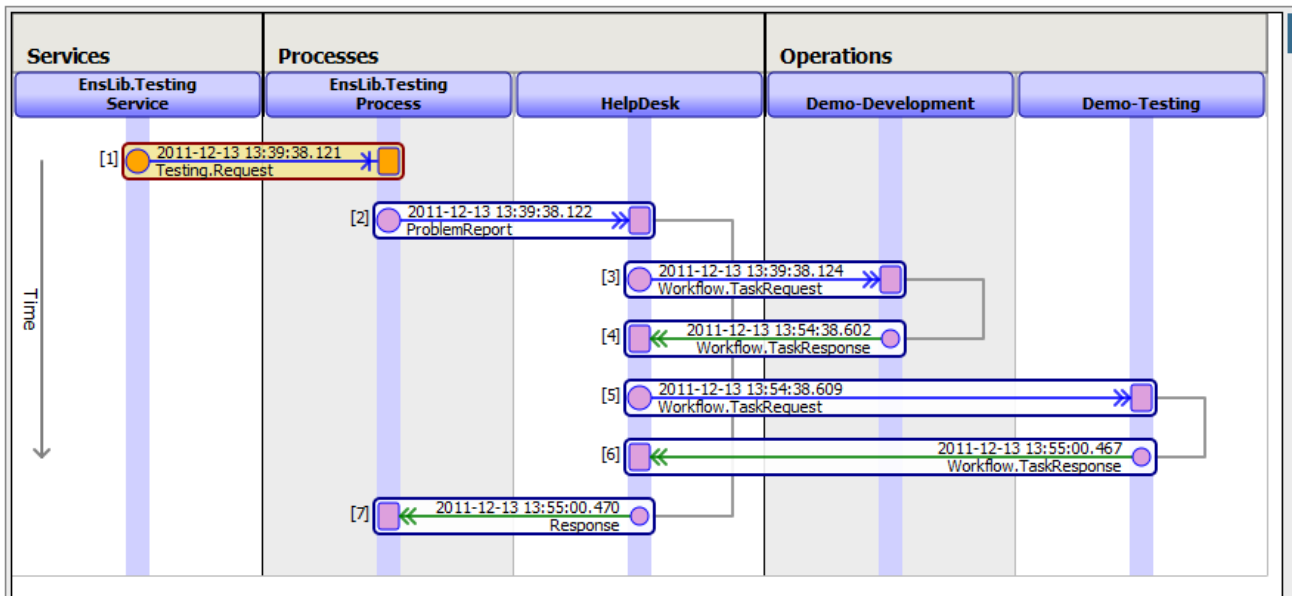
- Test task distribution.
- Test task cancellation.
- Test task escalation.
- Test exception cases.
- Test timeout behavior.
- View dashboards (if any).

You can monitor the activity via the Management Portal as described in "Managing Workflow Roles, Users, and Tasks" in *Managing Ensemble*. You can also use the Visual Trace, as discussed next.

4.2 Viewing Workflow Activity in the Visual Trace

All the statistical, maintenance, and display features that Ensemble supports for messages and business operations also apply to workflow tasks and workflow roles, respectively. These features include the Message Viewer, Message Purge, and the Visual Trace. For details, see *Managing Ensemble*.

Of particular interest, you can trace workflow task requests and responses with Visual Trace, as for other types of message. For example:



This example shows a session in which a task request was sent to the Demo-Development role and was addressed by a user in that role. Then the workflow process (HelpDesk) sent another task request to the Demo-Testing role. A user in that role addressed the request and the workflow process sent a response to the process EnsLib.Testing.Process.

For information on this example, see the appendix "Exploring the Workflow Sample."

A

Exploring the Workflow Sample

This appendix presents an sample that you can find in the package Demo.Workflow in the ENSDEMO namespace. So that you can more easily work with this sample, the appendix also includes some setup information and suggestions on points of interest. This appendix includes the following sections:

A.1 Overview of the Sample

The Demo.Workflow sample includes a business process called Demo.Workflow.HelpDesk. This accepts an incoming request that contains the basic information required for a problem report; that is, the name of the sender and a description of the problem. On receiving this request for action, HelpDesk begins a sequence of actions that continues until one of these actions resolves the problem.

HelpDesk first sends the problem report, as a task, to the development group. When a member of this group reports that the problem was solved, HelpDesk adds a note to the problem report and sends it, as a task, to the testing group. When a member of this group reports that the repairs were satisfactory, HelpDesk sends a final response to its caller, with a flag indicating that the reported problem was resolved.

A.2 Setup Tasks

To set up and work with this sample, do the following:

1. Create several users as described in "Users" in the *Caché Security Administration Guide*.
2. Configure each of these users as a workflow user, as described in "Managing Workflow Roles, Users, and Tasks" in *Managing Ensemble*.
3. In the ENSDEMO namespace, start the production Demo.Workflow.Production.

When you do so, Ensemble creates two workflow roles: Demo-Development and Demo-Testing. (It creates these roles because the **Auto Create Role** setting is enabled for the two workflow operations in the production.)

4. Assign the workflow users to these workflow roles, as described in "Managing Workflow Roles, Users, and Tasks" in *Managing Ensemble*.

   Some users can be in both roles.

5. Use the **[Ensemble] > [Testing Service]** page to send a few messages to HelpDesk:

   a. Click **Ensemble**, then click **Testing**, then click **Business Hosts**.

   b. For **Target Type**, click **Business Process**.

   c. For **Target Name**, click HelpDesk.

   d. Click **Test**.

   e. Type suitable values into the **ReportedBy** and **Problem** fields.

   f. Click **Invoke Testing Service**.

   Send several messages to make the scenario interesting.

6. Log into the DeepSee User Portal as each of the users and use the Workflow Inbox to manage the tasks.

   First log in as a user in the Demo-Development role and mark some tasks as complete; this generates additional testing tasks. Then log in as a user in the Demo-Testing role and mark the testing tasks complete.

   Tip: You might need to configure the /csp/ensdemo web application so that it accepts password authentication but does not accept unauthenticated access. For details, see in "Applications" in the *Caché Security Administration Guide*.

A.3 Sample Request Message

The following request message class defines the request message that comes into the HelpDesk sample business process. It represents a technical problem submitted by a customer. The request message class definition is as follows:

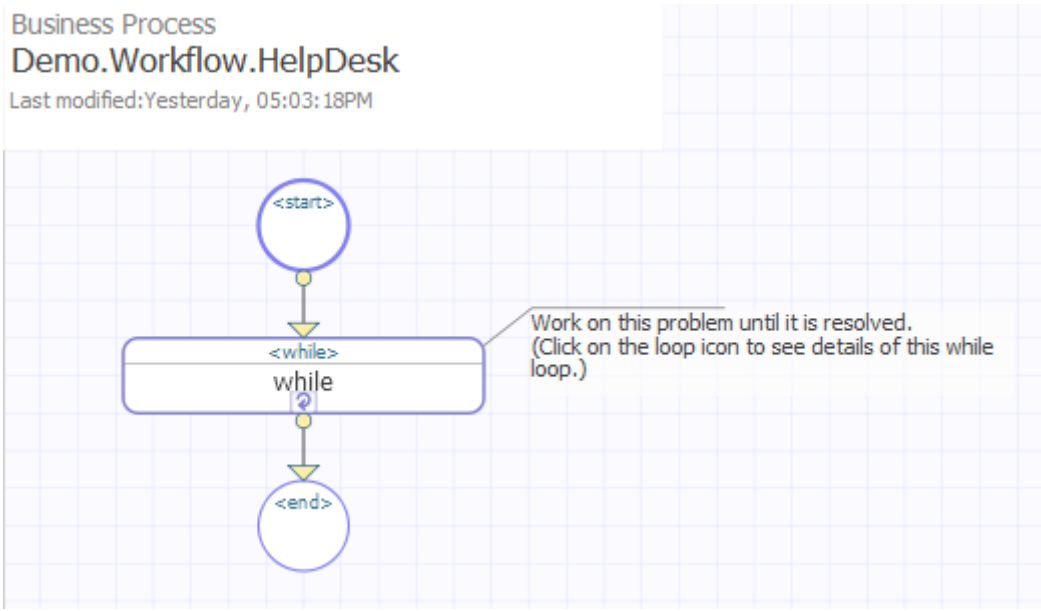
```
Class Demo.Workflow.ProblemReport
{
/// Name of customer reporting the problem.
Property ReportedBy As %String(MAXLEN = 60);

/// Description of the problem.
Property Problem As %String(MAXLEN = 200);
}
```

A.4 Sample Business Process Class

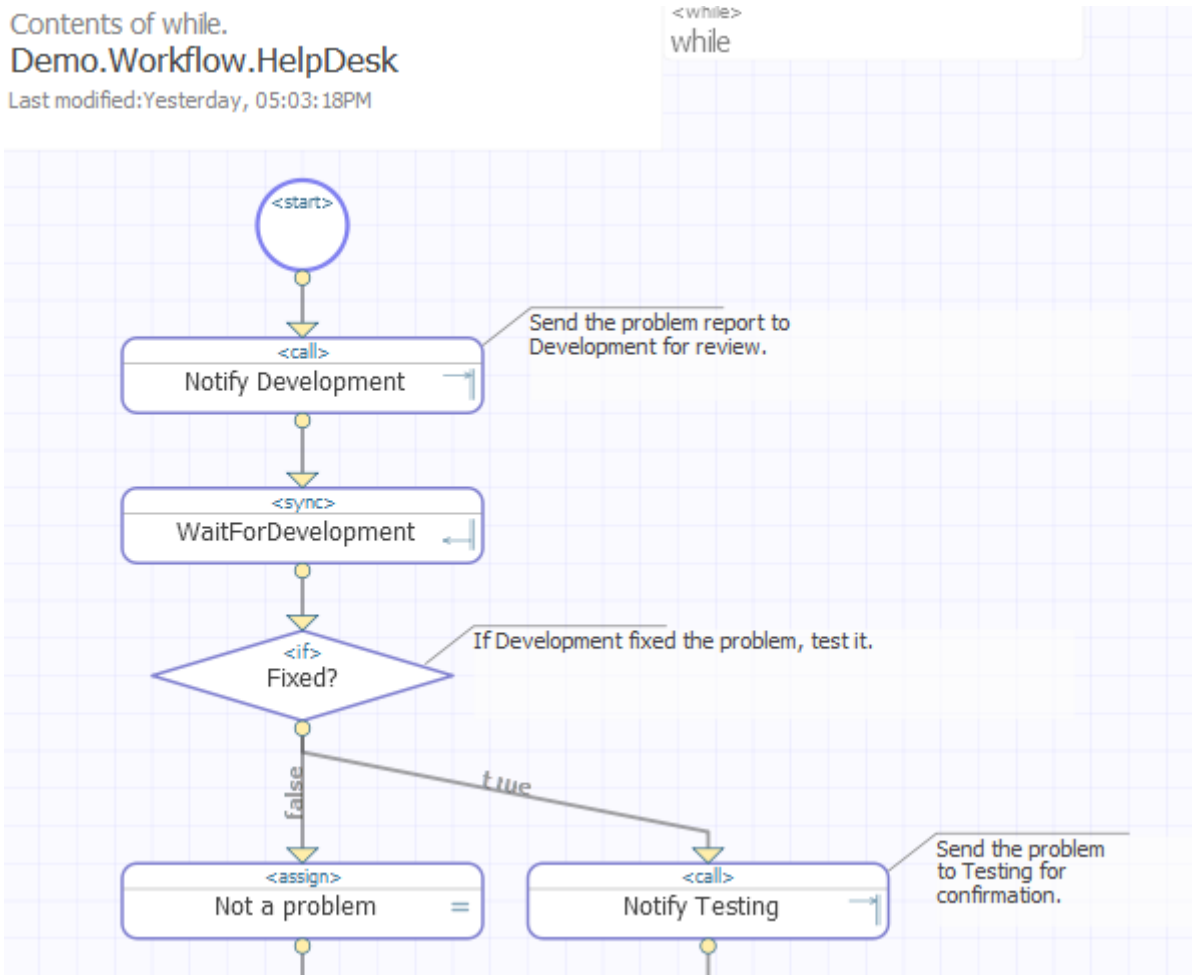
On receiving a Demo.Workflow.ProblemReport request message, the sample BPL business process Demo.Workflow.HelpDesk begins a sequence of actions that continues until one of these actions resolves the problem.

The HelpDesk business process consists of a while loop, during which a problem report is sent, first to the development group for a problem to be fixed, and next to the testing group for the fix to be tested. Upon completion of this sequence, the while condition "Resolved" is satisfied, and the business process ends, returning its final response to its caller.

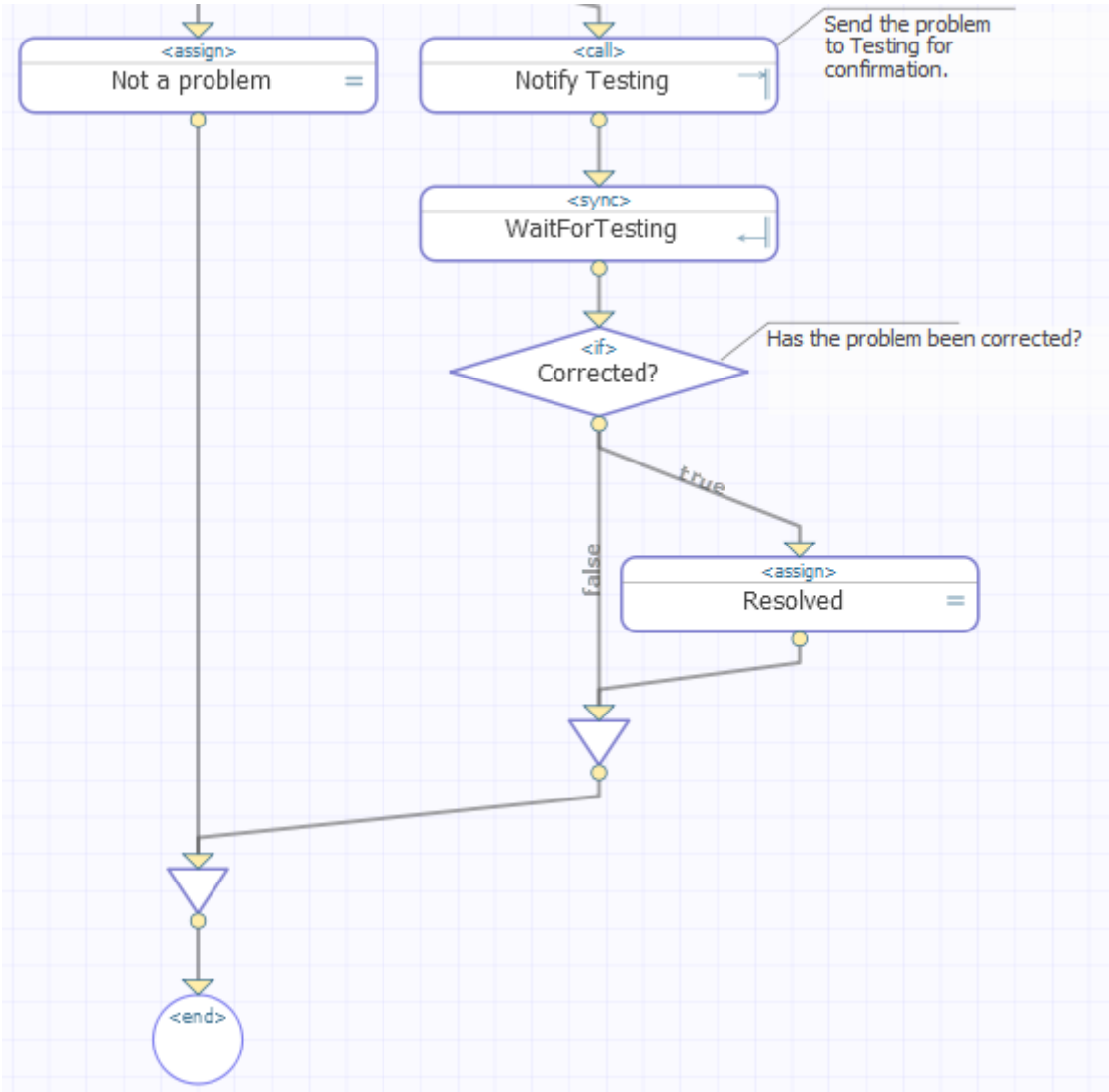
When viewed in the Business Process Designer, the HelpDesk business process looks like this:



The top area of the expanded while loop appears as follows:



The bottom section is as follows:



The code within the while loop makes two calls to workflow, using the built-in task request and task response classes for each call. The difference between the two <call> elements is easier to see in the BPL source code:

```
Class Demo.Workflow.HelpDesk Extends Ens.BusinessProcessBPL
{

XData BPL
{
<process request='Demo.Workflow.ProblemReport'
         response='Ens.Response' >
  <context>
    <property name='Resolved' type='%Boolean' initialexpression='0' />
    <property name='DevelopmentAction' type='%String' />
    <property name='TestingAction' type='%String' />
  </context>

  <sequence>
    <while name='while'
           condition='context.Resolved=0' >
      <annotation>
        Work on this problem until it is resolved.
        (Click on the loop icon to see details of this while loop.)
      </annotation>

      <call name='Notify Development'
            target='Demo-Development'
            async='1' >
        <annotation>
          Send the problem report to Development for review.
        </annotation>
```

```
        <request type='EnsLib.Workflow.TaskRequest' >
          <assign property='callrequest.%Actions'
                  value='"Corrected,Ignored"'
                  action='set' />
          <assign property='callrequest.%Subject'
                  value='"Problem reported by "_request.ReportedBy'
                  action='set' />
          <assign property='callrequest.%Message'
                  value='request.Problem'
                  action='set' />
          <assign property='callrequest.%FormFields'
                  value='"Comments"'
                  action='set' />
        </request>
        <response type='EnsLib.Workflow.TaskResponse' >
          <assign property='context.DevelopmentAction'
                  value='callresponse.%Action'
                  action='set' />
        </response>
      </call>

      <sync name='WaitForDevelopment' calls='Notify Development' type='all' />

      <if name='Fixed?' condition='context.DevelopmentAction="Corrected"' >
        <annotation>
          If Development fixed the problem, test it.
        </annotation>

        <true>
          <call name='Notify Testing'
                target='Demo-Testing'
                async='1' >
            <annotation>
              Send the problem to Testing for confirmation.
            </annotation>
            <request type='EnsLib.Workflow.TaskRequest' >
              <assign property='callrequest.%Actions'
                      value='"Corrected,Retest"'
                      action='set' />
              <assign property='callrequest.%Subject'
                      value='"Test this problem from "_request.ReportedBy'
                      action='set' />
              <assign property='callrequest.%Message'
                      value='request.Problem'
                      action='set' />
            </request>
            <response type='EnsLib.Workflow.TaskResponse' >
              <assign property='context.TestingAction'
                      value='callresponse.%Action'
                      action='set' />
            </response>
          </call>

          <sync name='WaitForTesting' calls='Notify Testing' type='all' />

          <if name='Corrected?' condition='context.TestingAction="Corrected"' >
            <annotation>Has the problem been corrected?</annotation>
            <true>
              <assign name='Resolved'
                      property='context.Resolved'
                      value='1'
                      action='set' />
            </true>
          </if>
        </true>

        <false>
          <assign name='Not a problem'
                  property='context.Resolved'
                  value='1'
                  action='set' />
        </false>

      </if>
    </while>
  </sequence>
</process>
}
}
```

A.5 Sample Control Flow

When the production is running, the BPL business process example Demo.Workflow.HelpDesk works as follows. Try comparing the steps in this control flow with the corresponding statements in the BPL source code, shown in the previous section:

1. A message of type Demo.Workflow.ProblemReport tells the business process who reported the problem, and provides a brief text string to describes the problem. The values for both these properties are provided by the business host that invokes this business process.
2. In preparing the <call> to the Demo-Development workflow operation, the Demo.Workflow.HelpDesk business process uses the incoming request properties ReportedBy and Problem to fill in the task request fields *%Subject* and *%Message*, respectively. Two other built-in task properties come into play as well: The call creates a list of possible user actions by assigning the *%Actions* field a value of `Corrected,Ignored`. The call also prepares a form field called `Comments` in which the person reviewing this task can enter data.
3. The business process makes the <call> to Demo-Development asynchronously, and provides a <sync> element to catch the task response.
4. The Workflow Engine associates the task with each workflow user in the Demo-Development workflow role.
5. One of the workflow users accepts the task. The Workflow Engine assigns the task to that user.
6. The assigned user edits the `Comments` field and clicks one of the actions **Corrected** or **Ignored**.
7. The task response returns. Its *%Action* field contains the value of the user action that completed the task (`Corrected` or `Ignored`). The business process saves this value into a execution context property called DevelopmentAction.
8. The business process uses an <if> element to test the DevelopmentAction value. The results are as follows:
   - When DevelopmentAction is `Corrected`, this value signals to the help desk business process that it should execute the <true> portion of its <if> element. The <true> element issues a <call> to the Demo-Testing workflow role as described in the next step.
   - When DevelopmentAction is not `Corrected`, this value signals to the help desk business process that it should execute the <false> portion of its <if> element, near the end of the BPL source code. Control reaches this statement when the DevelopmentAction is `Ignored` by the user. The business process returns its final response to its caller, commenting that the reported problem is `Not a problem` and setting the Boolean response value Resolved to 1 (true).
9. In preparing the <call> to the Demo-Testing workflow operation, the business process uses the original, incoming request properties ReportedBy and Problem to fill in the task request fields *%Subject* and *%Message*, respectively. However, unlike the previous <call>, the <call> to Demo-Testing has no form fields defined. Also, the list of possible user actions is different: *%Actions* is different: This <call> assigns the *%Actions* field a value of `Corrected,Retest`.
10. The business process makes the <call> to Demo-Testing asynchronously, and provides a <sync> element to catch the task response.
11. The Workflow Engine associates the task with each workflow user in the Demo-Testing workflow role.
12. One of the workflow users accepts the task. The Workflow Engine assigns the task to that user.
13. The assigned user clicks either **Corrected** or **Retest**.
14. The task response returns. Its *%Action* field contains the value of the user action that completed the task (`Corrected` or `Retest`). The business process saves this value into a execution context property called TestingAction.
15. The business process uses an <if> element to test the TestingAction value. The results are as follows:

- When TestingAction is `Corrected`, the business process returns its final response to its caller, commenting that the reported problem is `Resolved` and setting the Boolean response value Resolved to 1 (true).
- When TestingAction is not `Corrected`, the Boolean response value Resolved retains its initial value of 0 (false). The business process enters the top of the <while> loop again.

A.6 Dashboards and Metrics

To see a business metric and dashboard working together to display workflow role statistics, use Studio to view the sample classes Demo.Workflow.WFMetric and Demo.Workflow.Dashboard in the ENSDEMO namespace.

The following sample code from the business metric class Demo.Workflow.WFMetric calls three of the available statistical methods and assigns their return values to metric properties. The next appendix describes the available methods.

```
/// Sample business metric class for Workflow demo
Class Demo.Workflow.WFMetric Extends Ens.BusinessMetric
{

/// Active Tasks
Property ActiveTasks As Ens.DataType.Metric(AUTOHISTORY = 10, RANGELOWER = 0,
                                    RANGEUPPER = 50, UNITS = "Tasks")
                                    [ MultiDimensional ];

/// Active Load
Property Load As Ens.DataType.Metric(AUTOHISTORY = 10, RANGELOWER = 0,
                                    RANGEUPPER = 100, THRESHOLDUPPER = 90,
                                    UNITS = "%") [ MultiDimensional ];

/// Completed Tasks (since previous day)
Property CompletedTasks As Ens.DataType.Metric(AUTOHISTORY = 10, RANGELOWER = 0,
                                    RANGEUPPER = 100, UNITS = "Tasks")
                                    [ MultiDimensional ];

/// Calculate and update the set of metrics for this class
Method OnCalculateMetrics() As %Status
{
  // set the values of our metrics
  // %Instance is the current instance (RoleName in this case)
  Set tRole = ..%Instance

  Set ..ActiveTasks = ##class(EnsLib.Workflow.Engine).BamActiveTasks(tRole)
  Set ..Load = ##class(EnsLib.Workflow.Engine).BamActiveLoad(tRole)

  // Get task since start of previous day
  Set tStart = $ZDT($H-1,3)

  Set ..CompletedTasks =
      ##class(EnsLib.Workflow.Engine).BamCompletedTasks(tRole,tStart)

  Quit $$$OK
}

/// Set of instances for this metric class
/// There is one instance for every defined role.
Query MetricInstances() As %SQLQuery
{
  SELECT Name FROM EnsLib_Workflow.RoleDefinition
}

}
```

For information about the ObjectScript functions $ZDT ($ZDATETIME) and $H ($HOROLOG) used here , see the chapter "ObjectScript Functions" in the *Caché ObjectScript Reference*.

B

Available Workflow Metrics

The EnsLib.Workflow.Engine class provides methods that reporting statistics about workflow roles.

You can use these when you define business metrics, so that workflow progress can be tracked and analyzed by users. (For detailed instructions, see "Defining Business Metrics" in Developing Ensemble Productions. Also see "Exploring the Workflow Sample, "which presents an example.)

These methods are as follows:

BamActiveTasks()

```
ClassMethod BamActiveTasks(pRole As %String) As %Integer
```

Returns the number of active tasks for a given workflow role. The input argument *pRole* is the configured name of a workflow role. This method calculates the number of active tasks by getting the list of tasks assigned to this role and determining which of these tasks are not yet complete.

BamActiveLoad()

```
ClassMethod BamActiveLoad(pRole As %String) As %Integer
```

Returns a value indicating the active load for a given workflow role. The input argument *pRole* is the configured name of a workflow role. This method calculates the active load by getting the current number of active tasks for the workflow role, comparing it with the Capacity property from the workflow role definition, and presenting the result as a percentage of total capacity for the workflow role. The formula is:

```
(ActiveTasks/Capacity)*100
```

BamCompletedTasks()

```
ClassMethod BamCompletedTasks(pRole As %String,
                              pStart As %TimeStamp = "",
                              pEnd As %TimeStamp = "") As %Integer
```

Returns the number of completed tasks for a given workflow role, where:

- *pRole* is the configured name of a workflow role.
- *pStart* and *pEnd* are in %TimeStamp format: *yyyy-mm-dd*

This method calculates the number of completed tasks by getting the list of tasks assigned to this role and determining which of these tasks has a status of Complete. **BamCompletedTasks()** considers only those tasks that started within the time period specified by *pStart* and *pEnd*. The task may have completed after the *pEnd* time.

BamTasksWithStatus()

```
ClassMethod BamTasksWithStatus(pRole As %String,
                               pStatus As %String,
                               pStart As %TimeStamp = "",
                               pEnd As %TimeStamp = "") As %Integer
```

Returns the number of tasks that have a particular status for a given workflow role, where:

- *pRole* is the configured name of a workflow role.
- *pStatus* is one of the following strings indicating the status of the role:
  - `Unassigned`
  - `Assigned`
  - `Discarded`
  - `Cancelled`
  - `Completed`
- *pStart* and *pEnd* are in %TimeStamp format: *yyyy-mm-dd*

This method gets the list of tasks assigned to the role and determines how many of them currently have the given status. **BamTasksWithStatus()** considers only those tasks that started within the time period specified by *pStart* and *pEnd*. The task may have ended (for whatever reason) after the *pEnd* time.

If the status you are interested in is `Completed`, it is simpler to use **BamCompletedTasks()**.

BamAvgTaskTime()

```
ClassMethod BamAvgTaskTime(pRole As %String,
                           pStart As %TimeStamp = "",
                           pEnd As %TimeStamp = "") As %Integer
```

Returns the average duration (in seconds) of completed tasks for a given workflow role, where:

- *pRole* is the configured name of a workflow role.
- *pStart* and *pEnd* are in %TimeStamp format: *yyyy-mm-dd*

This method considers only those tasks that started within the time period specified by *pStart* and *pEnd*. The task may have completed after the *pEnd* time.

BamMinTaskTime()

```
ClassMethod BamMinTaskTime(pRole As %String,
                           pStart As %TimeStamp = "",
                           pEnd As %TimeStamp = "") As %Integer
```

Returns the minimum duration (in seconds) of completed tasks for a given workflow role, where:

- *pRole* is the configured name of a workflow role.
- *pStart* and *pEnd* are in %TimeStamp format: *yyyy-mm-dd*

This method considers only those tasks that started within the time period specified by *pStart* and *pEnd*. The task may have completed after the *pEnd* time.

BamMaxTaskTime()

```
ClassMethod BamMaxTaskTime(pRole As %String,
                           pStart As %TimeStamp = "",
                           pEnd As %TimeStamp = "") As %Integer
```

Returns the maximum duration (in seconds) of completed tasks for a given workflow role, where:

- *pRole* is the configured name of a workflow role.
- *pStart* and *pEnd* are in %TimeStamp format: *yyyy-mm-dd*

This method considers only those tasks that started within the time period specified by *pStart* and *pEnd*. The task may have completed after the *pEnd* time.