5. The System Architecture of a Reusable Workflow Management System

In this chapter, we introduce an object-oriented model, named "JOO-WfMS (**J**ava-based **O**bject-**O**riented **W**ork**f**low **M**anagement **S**ystem)", for workflow management system. In JOO-WfMS, we propose an object-oriented architecture and provide a class library for system developers and some GUI tools for process designers. Moreover, JOO-WfMS provide a run-time environment for workflow processes. JOO-WfMS adopts light-weight architecture. The developers can implement a simplest WfMS by implementing the basic components (Workflow engine, process and activity) listed in Figure 11. Depending on requirements, the advanced components can be added into system flexibly. Below, we introduce the software architecture of WfMS and details of JOO-WfMS.

5.1 The Stack of WfMS

Most WfMSs have complex architecture. An operation in WfMS may involve many components and mechanisms, e.g. the execution of an activity may involve user interaction, database manipulation, data transformation and failure recovery etc. Some of these components and mechanisms are provided in WfMS, others may require being implemented by system developers. Befor introducing JOO-WfMS, we define the stack of WfMS and classify the components within it. From the viewpoint of software development, we think this stack may bring up two benefits – (1) promote the reusability of all components in software, (2) increase the development efficiency of software. For example, to provide a database-independence design for persistence component, we separate the manipulation of database systems from the computation of workflow data. This design promotes the reusability of persistence component.

Moreover, the efficiency of software development can be increased because the system developers and process designers can concentrate on their own jobs. The stack of WfMS is illustrated in Figure 22:

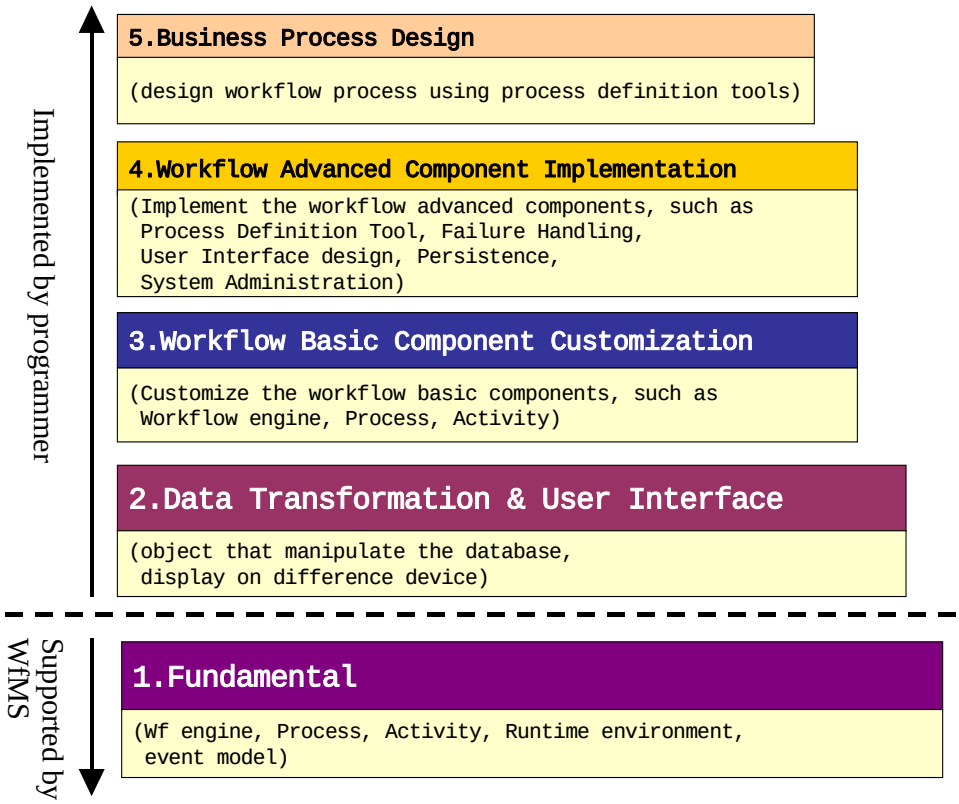


Figure 22: categorize the components of WfMS

1. **Fundamental**: the fundamental of WfMS provide basic components and mechanisms for system execution, developers may use these basic components to implement a customized WfMS. Components provided in this layer include the prototypes of components listed in Figure 11. Also, mechanisms like event handling or failure recovery should be provided in this layer.
2. **Data Transformation & User Interface Display**: components used for data transformation include the components for datastorage manipulation (e.g. the `DataSource` class in DAO) and data manipulation (e.g. the `ValueObject` class in DAO pattern). Most data of WfMS are kept in database, but some data

may be kept in other data storage like text files. The display protocols for different device should also be implemented in the components in this layer. The components in other layers may invoke the components provided here.

3. **Workflow Basic Component Customization & Implementation**: in this layer, the developers customize the basic components provided in fundamental according to requirements and implement the needed objects for these components. For instance, an interpreter may be implemented for workflow engine to interpret process defintions. Moreover, the task codes (execution code) for activities and the decision rules for transitions should be implemented and the user interfaces for activities should be designed in this layer.
4. **Workflow Advanced Component Customization & Implementation**: in this layer, the developers customize the advanced components provided in fundamental according to requirements and implement the needed objects for these components. Moreover, the window components and GUI editor for user interface are provided in this layer. The window components can use the user interface display components provided in layer 2 to display on different device.
5. **Business Process Desin**: the process designers design the workflow process according to business logic. The process designer may use the process definition tools to design workflow processes.

In general, we classify the components of WfMS into two categories, the fundamental components are provided by WfMS vendors and others are implemented or customized by programmers. After the introduction of stack of WfMS, we start to introduce the details of JOO-WfMS.

5.2 The System architecture of JOO-WfMS

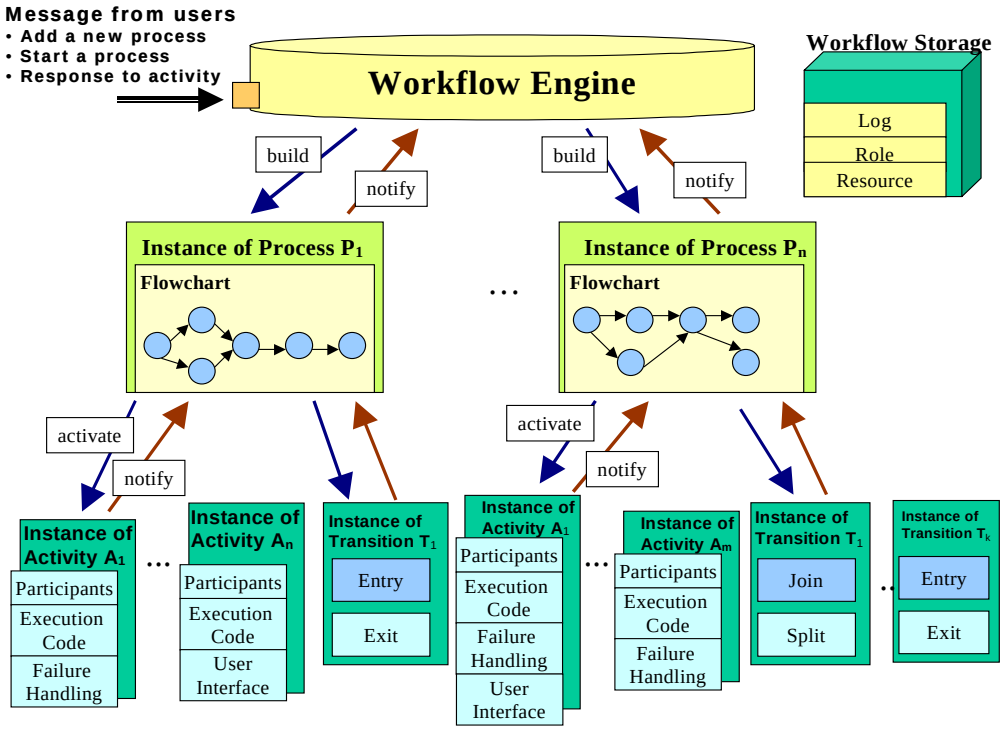


Figure 23: The system architecture of JOO-WfMS

Figure 23 illustrates the core and operational model of JOO-WfMS. The core of JOO-WfMS comprises the basic components – workflow engine, process and activity. As shown in Figure 23, there are several processe instances executing simultaneously and their executions are monitored by a workflow engine. The process instances are created by workflow engine, and a prcess instance will notify the workflow engine on the occurrences of events during execution. Several activity instances and transition instances are activated by a process instance. Similarly, an activity instance will notify the process instance on the occurrences of events during execution and so will a transition instance. Every activity comprises participants and execution code. The participant is the workflow user who participates in the execution of this activity. The execution code represents the task performed by this activity. Sometimes, user interface and failure handling components may be added into an activity. To load and

create activity and transition instances, we introduce the *flowchart* object in the process. When meeting branches during the run-time of a process, the flowchart and transitions will be used together to control the navigation of this process. In the following section, we will introduce the details of components of JOO-WfMS.

5.3 Workflow Engine

The workflow engine in JOO-WfMS is responsible for the following tasks:

- receive the messages from users
- interpret the process definition
- initiate the process and start it according to process definition.
- receive and deal with the events from processes.
- recover and resume process after system crash or the occurrence of failure.
- supervisory action for administration and monitoring purpose.

After started to run, workflow engine will wait the messages from users. Workflow users will send messages to workflow engine in the following three situations – (1) the process designer want to add a new process into system, (2) a user want to start a process, for instance, a student want to start a process to ask for leave. In this case, we call the user "*requester*". (3) a workflow user makes a response to the activity that he participates in.

In the first situation, the message from process designer may comprise the location of process definition and classes that will be executed in JOO-WfMS (e.g. the "task" program of activity). After receiving messages, workflow engine will do the following steps to add a new process into system:

A. get the unique identification (process definition id) of the process definition, the identification should be assigned by user (e.g. process designer), if not, the system may generate one for the process definition;

B. Interpret the process definition and generate a “flowchart” for process. Process definition written with any workflow model language should be supported in JOO-WfMS. But, the process definition written with XPDL may be the most suitable. The “flowchart” is used to record the information and execution states of all activities in the process. The process will load the activities that will be executed next according to flowchart. We will introduce flowchart at Section 5.4.

C. Perserve the flowchart of the process so that we can reuse it when the process is invoked. Currently, we store the flowchart in a hashtable and use the identification in step A as index.

In the second situation, the message from requester may comprise the identification of the process that will be started and some parameters for the execution of the process. After receiving the message, workflow engine will do some jobs to execute the specified process:

1. The workflow engine will first search the corresponding flowchart according to the identification and duplicate it
2. Create a process instance and generate an unique identification (process id) for it
3. assign the unique identification (process id), the duplicated flowchart and parameters to the instance
4. Start to execute the process.

In the final case, the message from the participant includes the information about the activity that he participates in and a activity and the data that the participant want to

send to the activity. After receiving the message, workflow engine will find the activity and assign the data to it.

In addition to handle the messages from users, workflow engine deals with the events from processes. In default, a process instance will send an event to workflow engine when its execution state changes. Programmers can assign workflow engine to execute some actions when receiving the events from processes, e.g. to save some information about the process.

Another responsibility of workflow engine is to recover and resume processes after system crash or the occurrence of failure. A workflow process may be executed for a long period, e.g. several months or several years. The execution of process may be broken off because of system crash or failure. The interrupted processes should be resumed after the system is rebooted. Workflow engine will execute the following actions to recover and resume processes:

1. Find out the unfinished processes according to the execution log.
2. Initiate process instances and assign the flowcharts to them according to the identifications of process definition.
3. Set the flowcharts according to the data of execution log.
4. Restart these processes.

5.3.1 Design Details

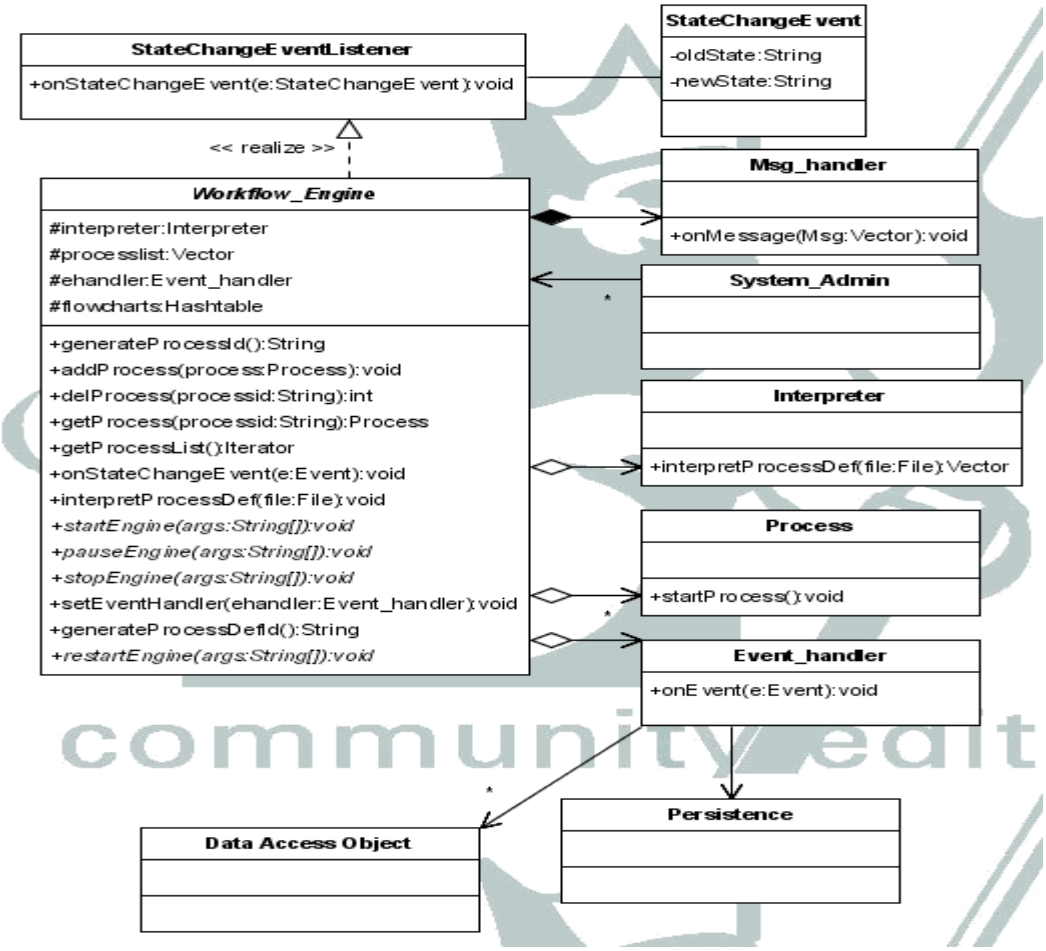


Figure 24: The class diagram of Workflow Engine

Figure 24 is the class diagram of workflow engine; it illustrates the important methods and the relationship between workflow engine and other classes.

`startEngine`, `pauseEngine`, `stopEngine` and `restartEngine` methods are used to control the execution of workflow engine. System administrators may control workflow engine via system administration tools, therefore these three methods are usually invoked by system administration tools. The `Workflow_Engine` class is an abstract class and these four methods should be implemented by developers. Now, there is a simple implementation for `Workflow_Engine` provided in JOO-WfMS.

`generateProcessDefId` method is invoked to generate an unique identification for a process definition. This method will be invoked when a new process definition is added into system. `generateProcessId` method will generate an unique identification for a process. Every executing process in JOO-WfMS has an unique process id, workflow engine will use process id to identify process.

Workflow engine is the listener of *StateChangeEvent*. When the execution state changes, process instance send a *StateChangeEvent* to inform workflow engine. The `onStateChangeEvent` method of `Workflow_Engine` is invoked to handle the *StateChangeEvent* from process. To promote the flexibility, the actions that deal with events are actually implemented in `Event_handler`. According to Figure 24, the current state and new state of the process instance that sent the event are recorded in *StateChangeEvent*

`InterpretProcessDef` method invoke the `InterpretProcessDef` methd of `Interpreter` to interpret process definitions, this method will return the flowchart of the process definition. The flowchart is a `Vector` which comprises many node objects.

`Msg_handler` is responsible for waiting the messages from users and dealing with them. When workflow engine is started, a `Msg_handler` instance is also started and runs as a TCP socket server. The `onMessage` method with a `Vector` as parameter is invoked when receiving messages from users. In fact, the `Msg_handler` plays an important role to handle the communication between workflow system and workflow users so the operation of `Msg_handler` involves some customized message protocols. We will have a detail introduction for these message protocols in

section5.5.3.2.

5.3.2 Discussion of Workflow Engine

To promote the reusability, we adopt a“componsitional design”for workflow engine; we separate the implementation of some functions from Workflow_Engine class. The separated functions are message handling, system administration and monitoring, process control, process definition interpretation and event handling. These functions are implemented in other clesses. Workflow engine can invoke them according to requirements. Therefore, we can change the invoked classes to meet the change of functional requirements instead of modifying `Workflow_Engine` class. Moreover, we use“Strategy pattern” [28] to design the relationship between Workflow_Engine and other classes. Using this pattern, the change of the algorithm or implementation of other classes will not affect `Workflow_Engine` class.

As mentioned before, we defined the reusability of workflow engine as that the workflow engine can be reused even if some of its functions need to be changed. We believe that the compositional design and strategy pattern support us to reach the goal.

But, if programmers want to change the way to control (e.g. start, stop or pause) workflow engine, they will need to extend the `Workflow_Engine` class and overwrite `startEngine`, `pauseEngine` and `stopEngine` methods.

5.4 Workflow Process

In JOO-WfMS, the workflow process component is responsible for the following tasks:

- Create the instances of activity and transition

- Start activities and transition,
- Receive and handle events from activities and transitions.
- Update the information of *flowchart*.

In this section, we will introduce *flowchart*, details about transition will be introduced later. The *flowchart* is designed for two purposes – (1) the information recorded in flowchart may be used for flow control purpose. (2) Process can create or destroy the instances of activity and transition according to the information in flowchart.

In workflow system, flow control is an important functionality. Flow control in workflow includes the following two jobs – to get the acitivities that will be executed next and to decide the navigation of process when meeting branch. However, the object-oriented systems lack a procedural representation of control flow [17]. The decomposition into classes typical of object-oriented architectures deemphasizes the control flow, distributing it among different objects. Thus, the global control flow and behavior are less visible than in procedural programs [8]. Therefore, an additional challenge of building object-oriented workflow architectures lies in *providing abstractions* that maintain an explicit representation of the control flow without violating the principles of good object-oriented design. In JOO-WfMS, programmers can define the" decision rules" in transition objects; workflow system will make decision for the navigation of process according to descision rules. But, how can process instance get the information about the successor activities and transitions to execute them? Flowchart is designed to solve this problem. That is, a process instance can get the information about successor activities and transitions from its flowchart.

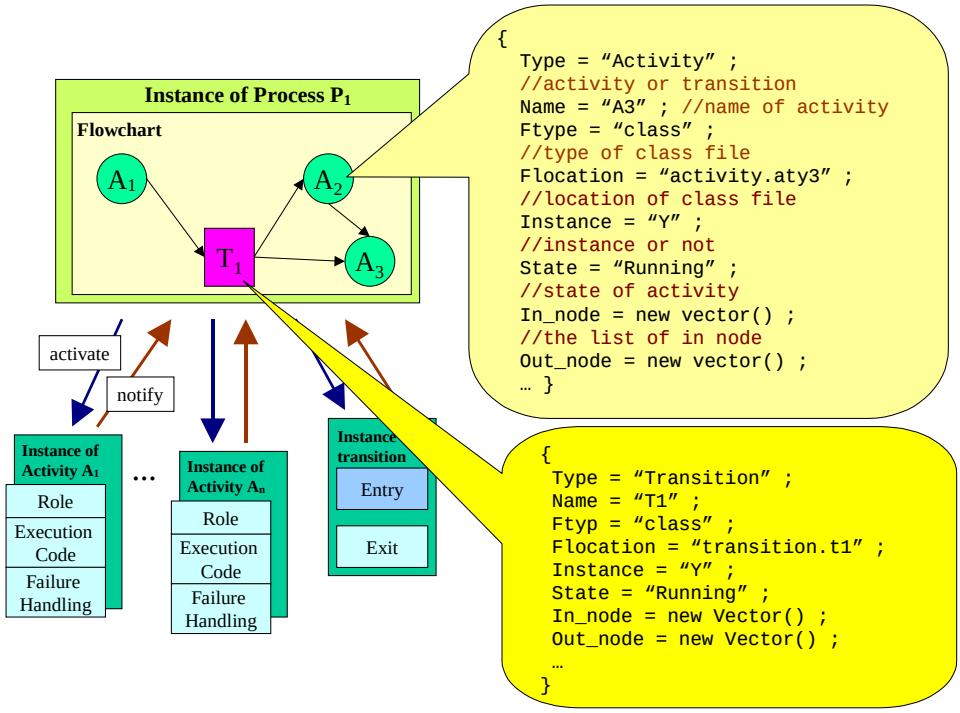


Figure 25: The sketch of Flowchart

Figure 25 shows the sketch of flowchart, flowchart comrises many *nodes* (we use a circular node to represent an activity, and use rectangle node for transition). The information about an acitivity or a transition are recoreded in a node. The fields of node are listed below:

- **Type**: type of the object that is represented by this node, the value of this field is either "activity" or "transition".
- **Name**: the name (or identification) of this activity or transition
- **Ftype**: the value of this field is either "class" or "ser". If the value is "class", it means that we can create a new activity instance or transition instance by using "new" operator. If the value is "ser", it means that the activity or transition is serialized into a local or remote file; we may load the serialized file from "Flocation" and deserialize it before instantiating.
- **Flocation**: the location of class file or serialized file.

- **Instance**: whether the activity or transition need to be instantiated before process start to exeucte or not. The value of this field is either “Y” or “N”.
- **State**: the execution state of the instance.
- **In_node**: references point to the nodes that represent the previous activities or transitions.
- **Out_node**: references point to the nodes that represent the successor activities or transitions.

After starting to execute, process will then start to execute activities according to flowchart. The starting activity of this process may be specified in flowchart. If the starting activity is not specified, the first activity of this process will be regared as the starting activity. After the completion of an activity, process will instantiate and execute next activities or transitions according to the information (Out_node) of flowchart. Also, tansition instance may use the information (State) of flowchart to make decision for the navigation of process.

Another tasks for workflow process is to receive and handle events from activities and transitions. An activity will send an event to process when its execution state changes and so does transition. When receiving this event, process instance will update the information (e.g. the state of activity) of flowchart. Programmers may assign process instance to execute some actions when receiving this event, e.g. to save some information about the activity or transition that sends this event. In addition, an activity will send an event to inform process of the occurrence of failure. When receiving this event, all activities will be stopped and some steps will be taken to recover the process.

5.4.1 Design Details

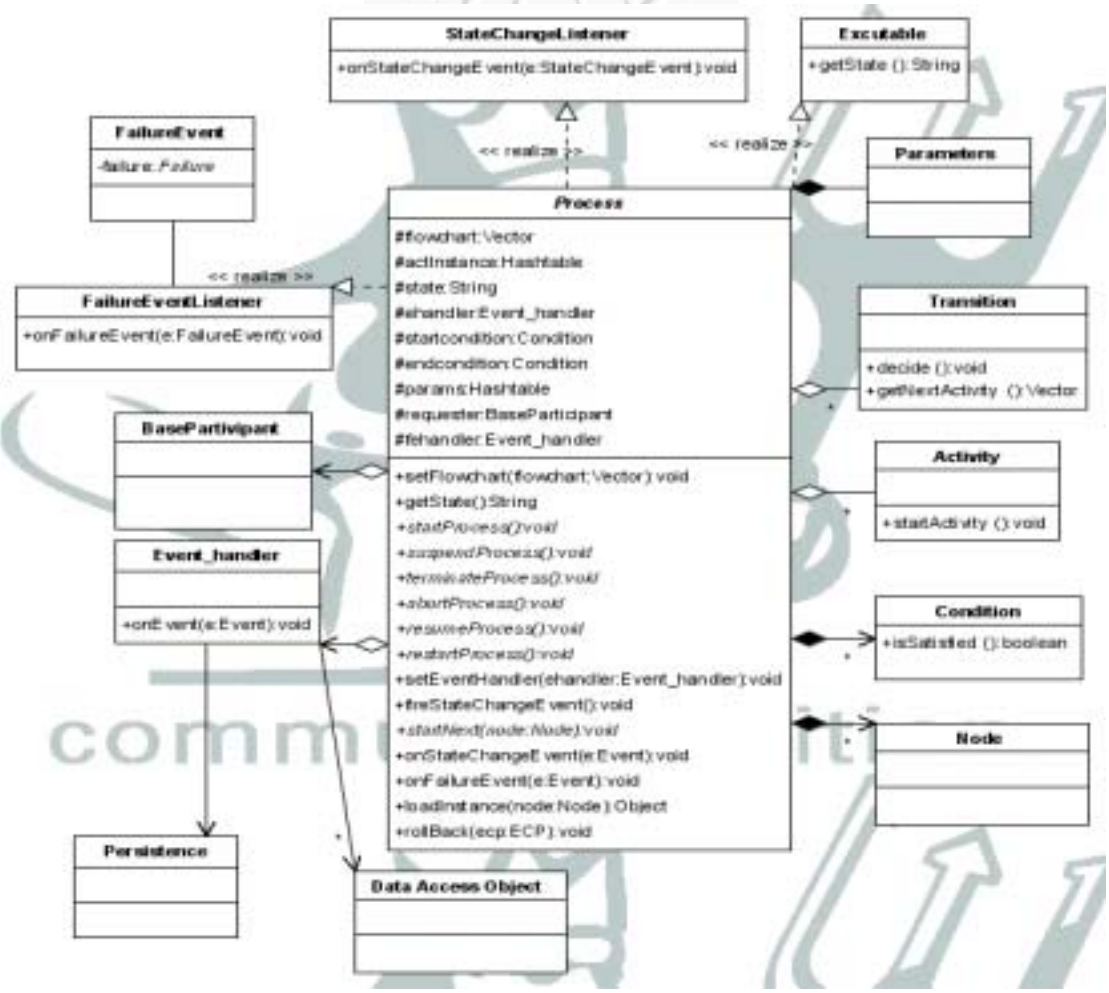


Figure 26: the class diagram of Workflow Process

Figure 26 is the class diagram of workflow process. Six methods are provided to control the execution of process instance - `startProcess`, `suspendProcess`, `terminateProcess`, `abortProcess`, `resumeProcess`, `restartProcess`. These methods may be invoked by workflow engine or system adminitration tools. These methods will also change the execution state of process instance. These method are abstract and therefore should be implemented by developers.

Figure 27 illustrates the state transitions for a process instance in JOO-WfMS. The ellipses represent the possible execution states of process instance, and text on arrows represents the invocation of metheds. System developers may extend it by add new states or modify the definitions of existing states.

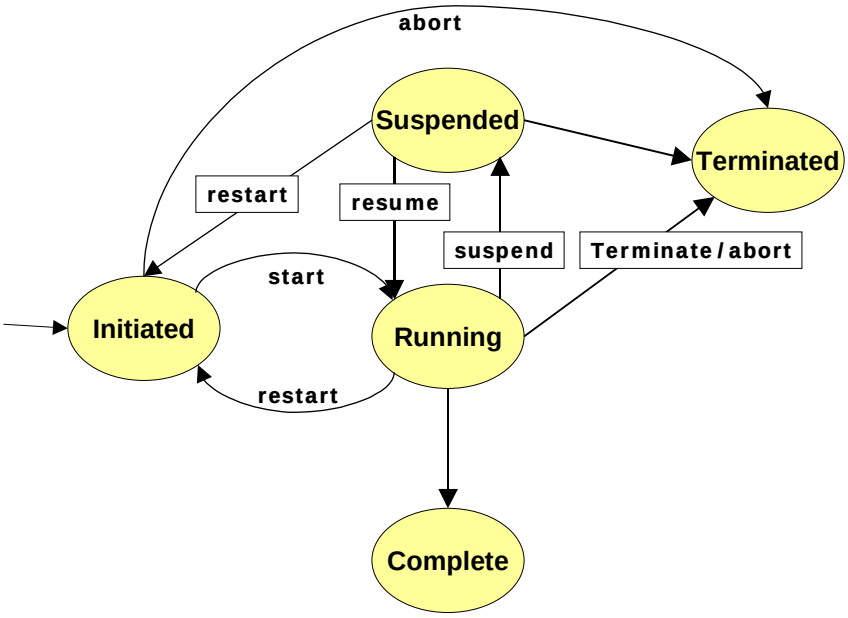


Figure 27: the state transitions for a process instance in JOO-WfMS

Refer to [9], we define these states as following:

- *Initiated* - the process instance has been instantiated, but has not been executed (the `startProcess` has not been invoked or the process has not fulfilled the conditions to cause it to start execution). The state of process will also be "initiated" after the `restartProcess` method is invoked.
- *Running* - the state of process will be "running" after the `startProcess` method or `resumeProcess` method is invoked.
- *Suspended* - the process will be suspended after the `suspendProcess` method is invoked. When the execution state is suspended, all the activities belong to

this process will be terminated and none of activities will be started. Moreover, when failure occurs, the process will also be suspended until `resumeProcess` method is called.

- *Terminated* - execution of the process instance has been stopped before its normal completion; any internal operations such as error logging or logging recovery data may be performed and the process instance destroyed. The state of process will be "Terminated" after the `terminateProcess` or `abortProcess` method is invoked.
- *Complete* - the process instance has fulfilled the conditions for completion;

Similarly, the activity and transition instances send a *StateChangeEvent* to process when their execution states change. The actions that deal with events are also implemented in `Event_handler`. In addition to the *StateChangeEvent*, an activity instance will send a *FailureEvent* to process instance. The *FailureEvent* includes a failure object, after receiving the *FailureEvent*, process will stop all executing activities and invoke the methods of this failure object for recovery.

The `startNext` method is used to start next activities or transitions. And the `loadInstance` method is invoked to load and create activity or transition instance according to the information in a node of flowchart. We will introduce the Failure object in section 5.5.1. The `startNext` method is abstract and should be implemented by system developers.

As mentioned before, every process may be invoked by a "request", the "requester" is represented with a `BaseParticipant` class. We will introduce the details of `BaseParticipant` class in section 5.7.

We use a `Hashtable` to keep the parameters of process. The parameters can be stored with the “(name, object)” form. The parameters of a process can be accessed by the activities and transitions that belong to this process.

5.4.2 Discussion of Workflow Process

Similar to the design of workflow engine component, we use a “compositional design” to promote the reusability of workflow process component. A process is combined with activities, transitions, event handler, and flowchart and so on. But, in addition to the“componsitional design”, we hope the process instance can load and instantiate activity and transition automatically according to process definition or other configuration files. The design of flowchart helps us to implement a mechanism for automation. With the support of Java Reflection, the instances of activity and transition can be automatically instantiated during the run-time of process according to the information recorded in flowchart. The flowchart of a process is generated by workflow engine, therefore, the process designer can change a process (e.g. change an activity) by modifing its process definition.

5.5 Workflow Activity

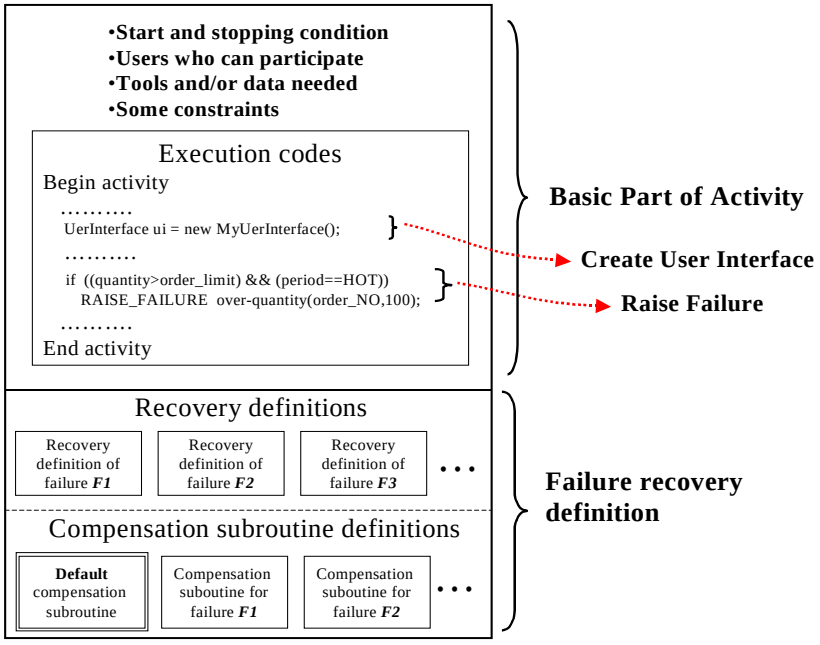


Figure 28: the structure of Activity

Figure 28 illustrates the structure of the activity in JOO-WfMS. The activity contains basic information for activity execution and additional failure-recovery definitions. The basic information for activity execution includes at least the starting and stopping conditions, the users who can participate, the tools and/or data needed to complete this activity, the constraints on how the activity should be completed (such as the time limits), and the *execution codes* of the activity. The additional failure-recovery definition in the activity for failure recovery comprises *recovery definitions* and definitions of the *compensation subroutines*. The execution code is the program that executes the activity and which may trigger the failure-recovery process when the execution causes a failure. In the execution code, a user interface may be invoked or create to interact with workflow users who participate in the execution of this activity.

5.5.1 Design Details

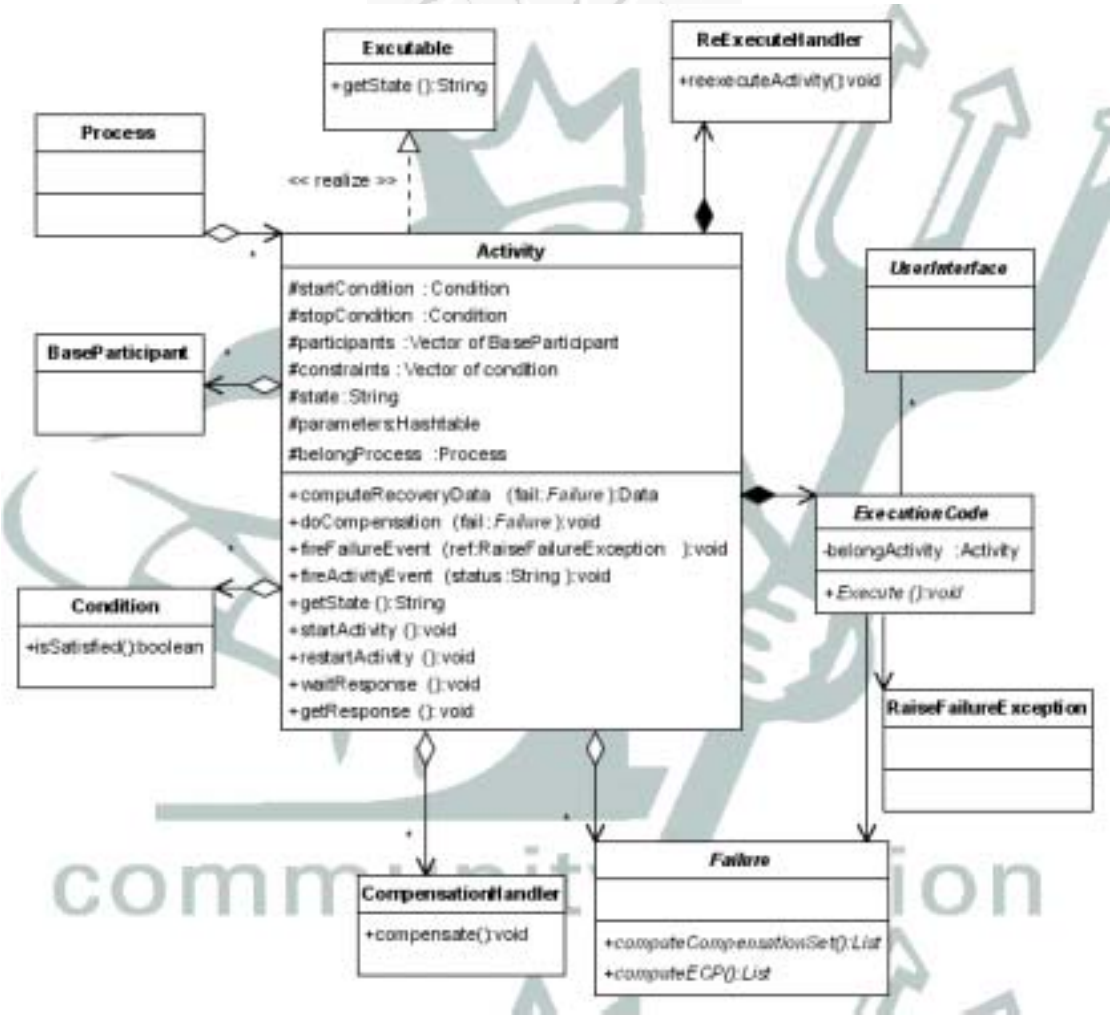


Figure 29: Class diagram of workflow activity

Figure 29 is the class diagram of workflow activity. Activity associates with several classes, including `Baseparticipant`, `Condition`, `ExecutionCode`, `Failure`, `CompensationHandler and ReExecuteHandler`. The "task" performed by the activity is implemented in the `Execute` method of the subclass of `ExecutionCode`. Programmer may raise failure or invoke the customized `UserInterface` to interact with user who participate the execution of this activity. We will introduce the way to raise failure and invoke the customized `UserInterface` later. The recovery definition, including how to compute compensation set and ECP of the failure, is implemented within the subclass of

`Failure`. The way to compensate the acitivity after the occurrence of failure is implemented within the class that implements `CompensationHandler` interface. After compensation, the class that implements `ReExecuteHandler` will be invoked to reexecute this activity.

Similar to Process instance, the Activity instance changes its state when `startActivity`, `restartActivity`, `waitResponse` and `getResponse` methods are invoked. The state transitions for activity instance are shown in Figure 30.

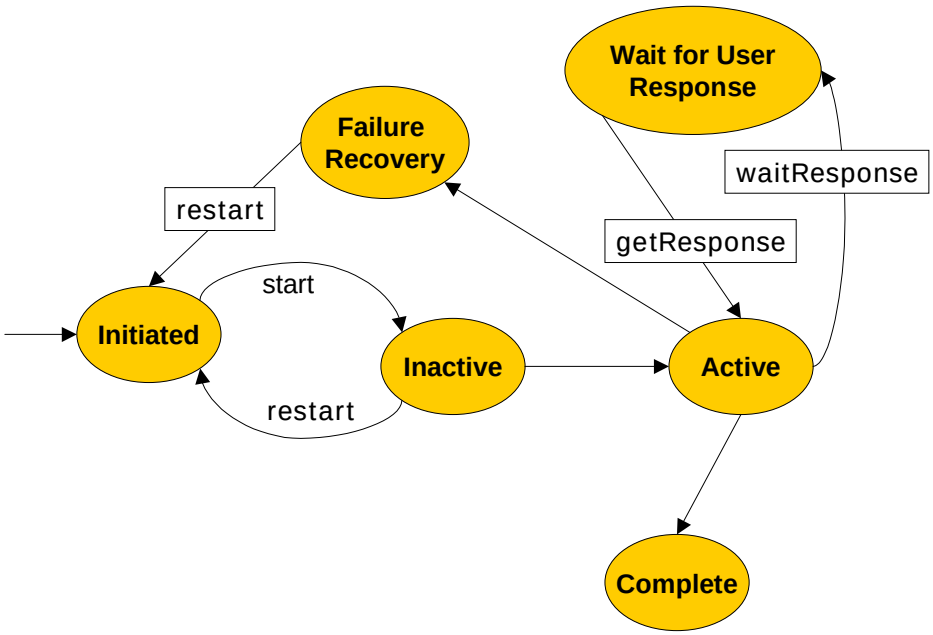


Figure 30: The state transitions for activity instance in JOO-WfMS

Refer to [9], we define the meaning of every state below:

- *Initiated* - the activity within the process instance has been created but has not yet been started. The state will also be “initiated” after the restart method is invoked.

- *Inactive* – startActivity method has been invoked, but the “task” has not yet been executed (because activity starting conditions have not been met)
- *Active* – the state of activity will change from “inactive” to “active” automatically when the strating conditions are met.
- *Failure recovery* – the state will change from “active” to “failure recovery” automically when failure occurs. After finishing the steps of recovery, restart method will be invoked.
- *Wait for user response* – the execution of the activity will be suspended to wait response from user after the invocation of waitResponse method. After getting response, system will invoke getRespose method to continue the execution of this activity.
- *Completed* - execution of the activity instance has completed (and any stopping conditions will be evaluated)

5.5.2 Discussion of Workflow Activity

A key point of the design is to put the “task program” apart from activity. As mentioned before, there may be many different “task program” in a business process but the ways that WfMS executes activities are similar. Moreover, the “task program” is implemented by the programmer who is responible for business process implementation; he just needs to concentrate his attention on the implementation of business logic, and doesn’t need to care how system executes activity. In a simple case, the programmer only needs to inherit ExecutionCode and implements “task program” in the Execute method, and then, assigns the customized ExecutionCode to Activity. Similarly, we separate the parts that change frequently from activity (e.g. the starting and stopping conditions, classes related to failure recovery, etc) and the activity uses them by manipulating interface or abstract class. Therefore, the change

of the implementation of these parts will not influence the implementation of activity.

However, if the programmers want to change the details of the operation of the activity, for example, changes the code of startActivity method, they must inherit the activity class and overwrite the related method.

5.5.3 User Interface

The synergy between humans and software represents one of the key features of workflow [8][19]. Some tasks in a process may need human users to participate in. Most WfMS products have their own mechanisms to support the intervention of users. For instance, [8][20][21] provide “worklist” to keep the “workitems” for human users. Some WfMSs may invoke extrinsic components or systems to handle the intervention of users, e.g. the user communication mechanism in Woorks [43] is supported by an extrinsic system called UTUCS. Figure 31 sketches how the worklist component extends the framework to support human workers in Micro-workflow [8]. The tasks (domain object) that do not need to invelove human users to execute are executed by process component and the tasks that involve human users are handled by worklist component.

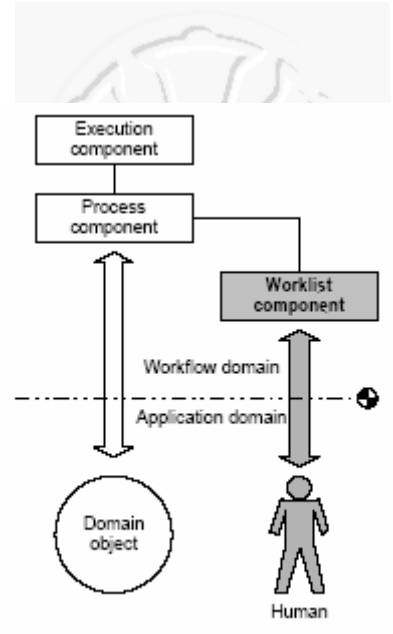


Figure 31: the worklist component adds to the Micro-Workflow framework to support human workers.

WfMC' defines the functions of Workflow Client Application in the interface 2 of workflow reference model. Figure 32 illustrates the interface 2 (client application interface) of WfMC's reference model.

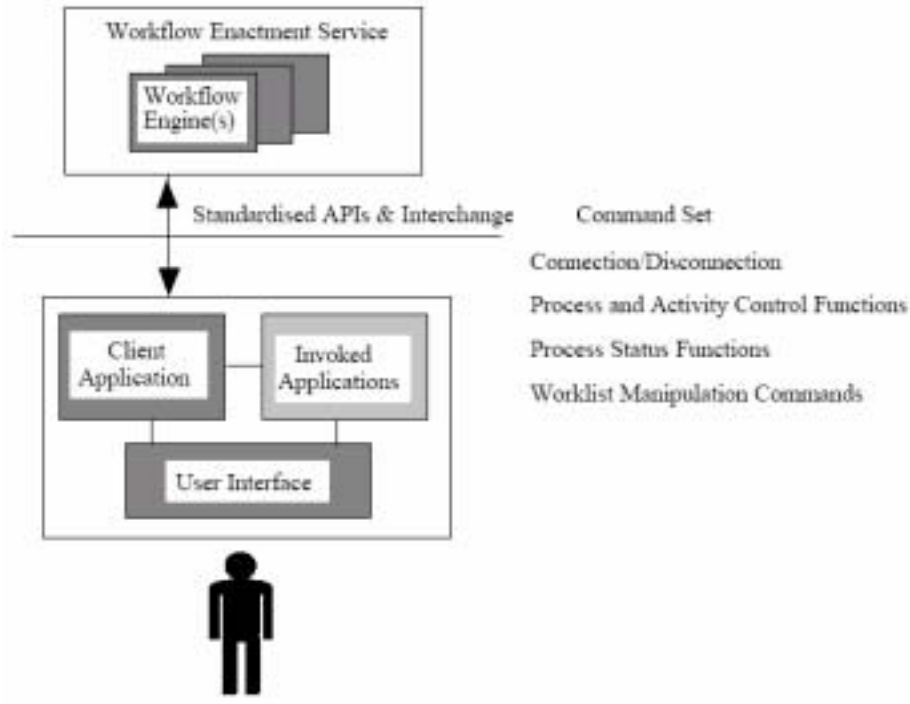


Figure 32: Client Application Interface

In this part, JOO-WfMS refers to [8] [9] but we do not completely obey these designs. In JOO-WfMS, we call the components related to user communication”User Communication Components (UCC)”.

Interacting with human users includes two jobs – (1) show messages to participants, (2) deliver user responses to system. Human users interact with system via the human-computer interface. System will show messages (text, message box or form) in *human-computer interface* and the participants may make responses according to these message. The human-computer interface equals to the user interface mentioned before.

The user response handling mechanism is important to user communication mechanism in JOO-WfMS. In most case, the participants may not deal with the “workitems” assigned by system and make response immediately. Therefore, we need a mechanism to allow human users to work together with system in an “asynchronous” way. That is, the user response handling mechanism in JOO-WfMS supports users to work in “asynchronous” way. Below, we will introduce the designs of human-computer interface and user response handling mechanism in JOO-WfMS.

5.5.3.1 Human-Computer Interface

Recently, more and more applications are based on web technologies (e.g. Java Applet, Java Server Pages [24], Active X and Active Server Pages…etc). User may access these applications via web browsers. Currently, we use Java Servlet and Java Server Pages to implement the human-computer interface in JOO-WfMS. But when designing and implementing the web-based human-computer interface, we meet a problem – to implement different user interfaces for different activities are laborious.

Every activity that requires human user participating in needs a human-compuer interface with different contents. If there are ten activities in a process need human users to participate in, developers have to write ten different JSP or servlet programs. Moreover, as we mentioned in Section 4.3.1, the interlacing of HTML tags and JSP expression make us difficult to implement GUI editor and this code is hard to be reused for different device.

To solve these two problems, we refer to the windows programming in Java to implement the human-computer interface in JOO-WfMS. The programmers can build a graphic user interface by using or entending the classes (e.g. Button, TextField…etc) in `java.awt` or `java.swing` package. There are two benefits for programmers to use the classes provided in Java: (1) the programmers can use GUI tools to build user interface; there are many products providing GUI tools for Java programmers such as Borland Jbuilder, IBM VisualAge for Java; (2) the same programs can be reused on different platforms. The classes in in `java.awt` or `java.swing` package are abstract representation of window components. In fact, the implementations for the displays of these components on different platforms (e.g. Windows, linux) are different. Because these differences are handled by Java Vitual Mechine (JVM), the programs can be written once and used everywhere. Therefore, similar to the way of window programming in Java, we provide some window components for developers to build human-computer interfaces for different activities and design a mechanism to handle the displays on different platflorms.We call the window components provided in JOO-WfMS "*User Interface Component (UIC)*"。 The functions and attributes of window components provided in JOO-WfMS are listed in Table 1.

UIC Component List

Container: component that contain some sub-items.	
1. Page: the root component of all other UIC components in a display. In a display, all User Interface Components are contained in page component, and every display has only one page component.	
Attributes	1. **Caption** – the title of html page 2. **Background image** – the background image of html page 3. **Background color** – the background color of html page 4. **Program** – some program written with JavaScript or VBScript to do some actions (e.g. pop up some messages) before the page is loaded.
2. Radio Button Group: the container contains some radio buttons, in radio button group, only one radio button can be "checked".	
Attributes	1. **Caption** – the caption of this Radio Button Group 2. **Name** – the name of this Radio Button Group
3. Check box Group: the container contains some check boxes, in check box group; more one check box can be "checked".	
Attribute	1. **Caption**– the caption of this Check box Group 2. **Name** – the name of this Check box Group
4. Pull-down Manu: a container that contains some "selectable items", more than one item can be selected if the "multiple" attribute is true.	
Attributes	1. **Caption** - the caption of this Pull-down Manu 2. **Name** – the name of the Pull-down Manu 3. **Multiple** – if the value of this attribute is true, more than one item can be selected.
5. Form	
Attributes	1. **Name** – the name of the form 2. **Method** – how the form will be send to target, either "GET" or "POST" will be the value of this attribute, usually. 3. **Action** – the target where the form will be submitted to. 4. **Program** - some program written with JavaScript or VBScript to do some action (e.g. pop up some messages) before submitting.
Item: the UIC components that do not contain another UIC component as sub-item.	
InputableItem: the UIC components that user can input or select some value in.	
1. TextField: the component that allow users to input texts, numbers and so on in a single line.	
Attributes	1. **Caption** – the caption of the textfield 2. **Name** – the name of the textfield 3. **Value** – the value of the textfield

	4. **Size** – the display size of the textfield 5. **Type** – the type of the textfield, the type can be either text or password.
2. TextArea: the component that allow users to input texts, numbers and so on in multiple lines.	
Attributes	1. **Caption** – the caption of the textarea 2. **Name** – the name of the textarea 3. **Value** – the value of the textarea 4. **Cols** – the number of columns of the textarea 5. **Rows** – the number of rows of the textarea
3. File Upload: the component that allows users to upload files to server.	
Attributes	1. **Caption** – the caption of the file upload component 2. **Name** – the name of the file upload component 3. **Value** – the value of the file upload component 4. **Size** – the display size of the file upload component
Non-InputableItem: items that just display something.	
4. Radio Item: the selectable component contained in radio button group.	
Attributes	1. **Caption** – the caption of the radio item 2. **Name** – the name of the radio item 3. **Value** – the value of the radio item 4. **Checked** – the “Checked” is true if the radio item is selected.
5. Check Box Item: the selectable component contained in check box group.	
Attributes	1. **Caption** – the caption of the check box item 2. **Name** – the name of the check box item 3. **Value** – the value of the check box item 4. **Checked** – the “Checked” is true if the check box item is selected
6. Select Item: the selectable component contained in pull-down menu.	
Attributes	1. **Caption** – the caption of the select item 2. **Name** – the name of the select item 3. **Value** – the value of the select item 4. **Selected** – the “Checked” is true if the select item is selected
7. Image: component that display an image file on the screen.	
Attributes	1. **Caption** – the caption of the Image 2. **Name** – the name of the Image 3. **Width** – the width of the image when displayed. 4. **Height** – the height of the image when displayed. 5. **Value** – the file path of the image on server
8. Label: component that display a sentence of text.	

Attributes	1. **Caption** – the caption shown in front of the text. 2. **Value** – the text that will be shown 3. **Foreground color** – the foreground color of the text 4. **Background color** – the background color of the text 5. **Font** – the font of the text 6. **Size** – the size of the text
9. Link: component that represent a hyperlink target	
Attributes	1. **Caption** – the caption of the hyperlink. 2. **Value** – the target document of this hyperlink. 3. **Font** – the font of this hyperlink. 4. **Target window** – the window where the target document shows. 5. **Size** – the size of the text
10. Button	
Attributes	1. **Name** – the name of the button 2. **Value** – the text shown on the button 3. **Type** – the type of this button, it can be "Submit" or "Reset" or "Button".

Table 1: The list of User Interface Components in JOO-WfMS

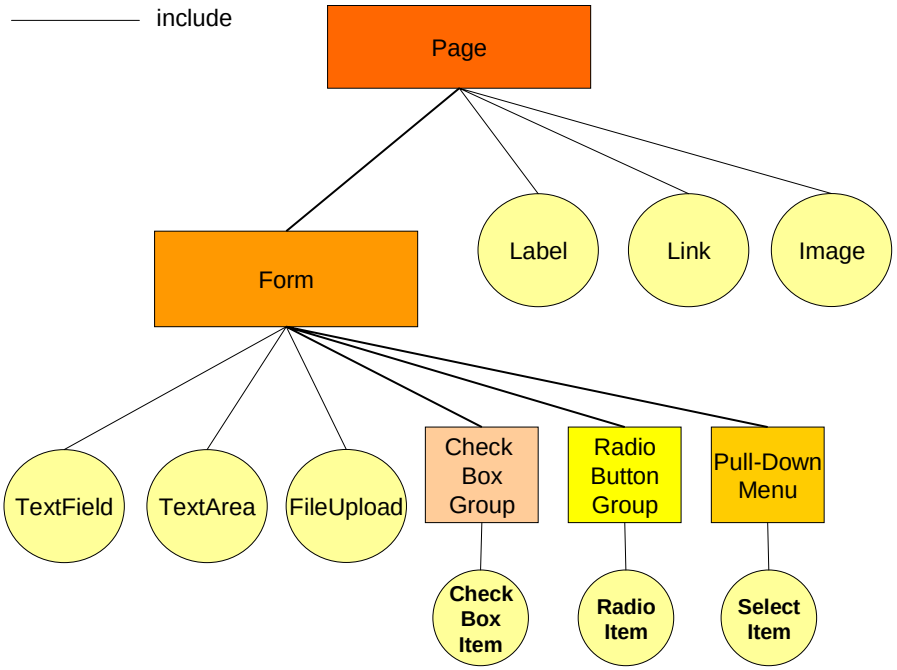


Figure 33: Relationship between User Interface Components

Figure 33 illustrates the relationship between UICs, these UICs can be classified to

two categories – *Container* and *Item*. A *Container* may have other *Containers* or *Items* as sub-items, whereas an item cannot have any subitem. The *Items* can be classified to two types – *InputableItem* and *Non-InputableItem*. An *InputableItem* (e.g. TextField, TexArea) allows users to input text or number, but a *Non-InputableItem* is only for displaying a message. There are several limitations for using these UICs:

1. Except for *Radio Item*, *Select Item*, *Check Box Item* and *Page*, a *Page* can include all kinds of *Containers* and *Items* as sub-items.
2. A *Form* can include all kinds of *Containers* and *Items* as sub-items except for *Radio Item*, *Select Item*, *Check Box Item*, *Page* and *Form*.
3. A *Radio Button Group* can only include *Radio Items*.
4. A *Check Box Group* can only include *Check Box Items*.
5. A *Pull-down Menu* can only comprise *Select Items*.
6. There can be only one *Page* in a Human-Computer Interface and this *Page* should be root, as shown in Figure 33.
7. The *InputableItems* that allow user to input value and return value to WfMS should be included in a *Form*.

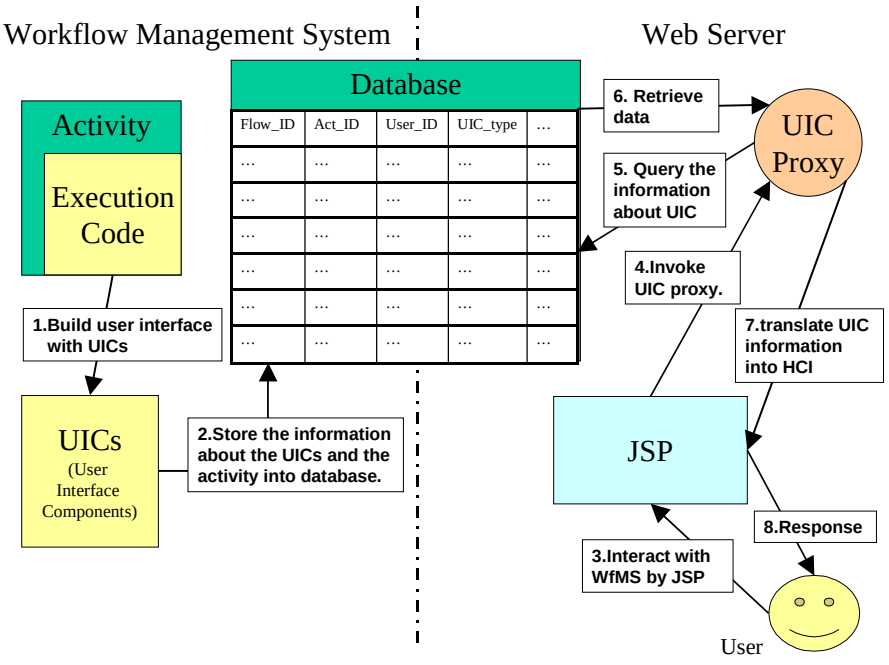


Figure 34: The Operation Model of Human-Computer Interface in JOO-WfMS.

The operation model of Human-Computer Interface in JOO-WfMS is shown in Figure 34 and we introduce the steps below:

1. At first, the developers will invoke a user interface which has been built or use UICs to combine a new user interface in the Execution Code of an activity.
2. Setting attributes for every UICs in this user interface. These attributes will be stored in the UCC table in database. The schema of UCC table is shown in Table 2:

DataBase Schema: UIC					
	Field name	Type	Length	Description	Allow Null
1.	Pid	String	20	The id of the process that the user interface belongs to.	No
2.	Aid	String	20	The id of the activity that the user interface belongs to.	No
3.	Uid	String	50	The id of the user who will access this user interface	No
4.	c_type	String	10	The type of this UIC (Page, Form, Label or Link … etc)	No

5.	c_name	String	25	The name of the UIC	No
6.	c_value	String	255	The value of the UIC	Yes
7.	c_font	String	50	The font used to display the messages in this UIC.	Yes
8.	c_caption	String	50	The caption of this UIC	Yes
9.	c_size	Int	--	The size of this UIC	Yes
10.	c_cols	int	--	The number of columns of this UIC (for *TextArea* component)	Yes
11.	c_rows	int	--	The number of rows of this UIC (for *TextArea* component)	Yes
12.	c_width	int	--	The width of this UIC	Yes
13.	c_height	int	--	The height of this UIC	Yes
14.	c_fgcolor	String	10	The foreground color of this UIC	Yes
15.	c_bgcolor	String	10	The background color of this UIC	Yes
16.	c_bgimage	Binary	--	The background image of this UIC	Yes
17.	c_imgvalue	Binary	--	The value of an *Image* component	Yes
18.	c_checked	Boolean	1	Default selected if this value is true (For *Radio Item* and *Checkbox Item*)	Yes
19.	c_chkvalreq	Boolean	1	Wheather the value of this UIC need to be checked or not.	Yes
20.	c_chkprag	String	--	Programs used to check the value of this UIC	Yes
21.	c_parent	String	255	The name of the parent of this UIC	Yes
22.	c_extattr	String	--	The extended attributes specisfied by developers	Yes
Primary keys: Pid, Aid and c_name					

Table 2: The database schema of UIC table

3. An user send a request to a specific JSP via web browser;

4. After receiving the request, the JSP will invoke UIC_Proxy to retrieve data and translate data of UICs in this user interface.
5. The UIC_Proxy queries the data of UICs form database.
6. The database returns the results of querying.
7. UIC proxy translates the data of UICs into a specific format (e.g. HTML tags).

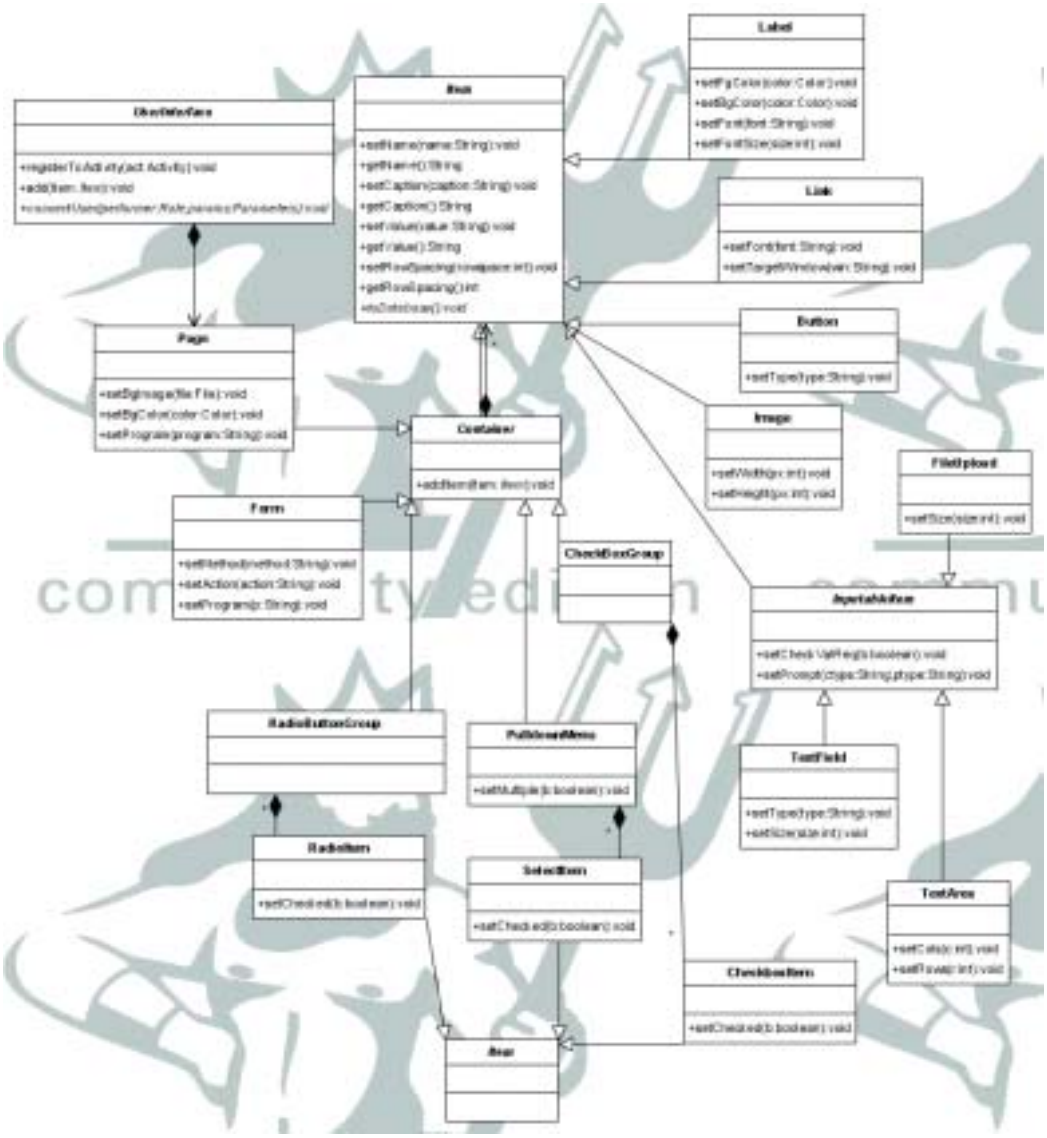


Figure 35: The class diagram of User Interface Components

Figure 35 is the class diagram of UICs. All UICs are the subclasses of `Item`. According to Table 1, an Item is non-inputable. Therefore, the subclasses (e.g. `Image`,

`Link` and `Label`) are non-inputable. The UICs that allow inputting should be the subclasses of `InputableItem`. The `InputableItem` is also the subclass of `Item`. Two advanced methods are defined in `InputableItem` – `setCheckValReq` and `setPrompt`. The former is used to set wheather the value of this UIC needs to be checked or not and the latter is invoked to set the checking type (alert or confirm) and the pop-up messages when the vale is invalid (e.g. the length of the value is more than a specific number). We use the Composite Pattern [28] to design the relationship between UICs; the design allows us to trace all the sub-items of a Page by using recursion.

In ExecutionCode, the programmer invokes the subclass of UserInterface to interact with User, the snatch of `Execute` method in ExecutionCode class is shown in Figure 36:

```
Page page = new Page (“show message”);
… //set the attributes of page
Label label = new Label (“Message”);
… //set the attributes of label
UserInterface myui = new UserInterface ();
myui.add (page);
myui.registerToActivity (belongActivity);
myui.connectUser (performer, parameters);
belongActivity.waitResponse ();
…
```

Figure 36: The code to create user interface in the execution code of an Activity in JOO-WfMS

Another important component in Human-Computer Interface is the UIC Proxy. As mentationed before, the UICs are the abastract representation of window components and their attributes and contents will be stored in database. The task of UIC_Proxy is to retrieve the data of UICs from database and to translate these data into a specific

format. Figure 37 illustrates a translating example. In this example, the UIC_Proxy translates the data of Label and TextField components into HTML tags.

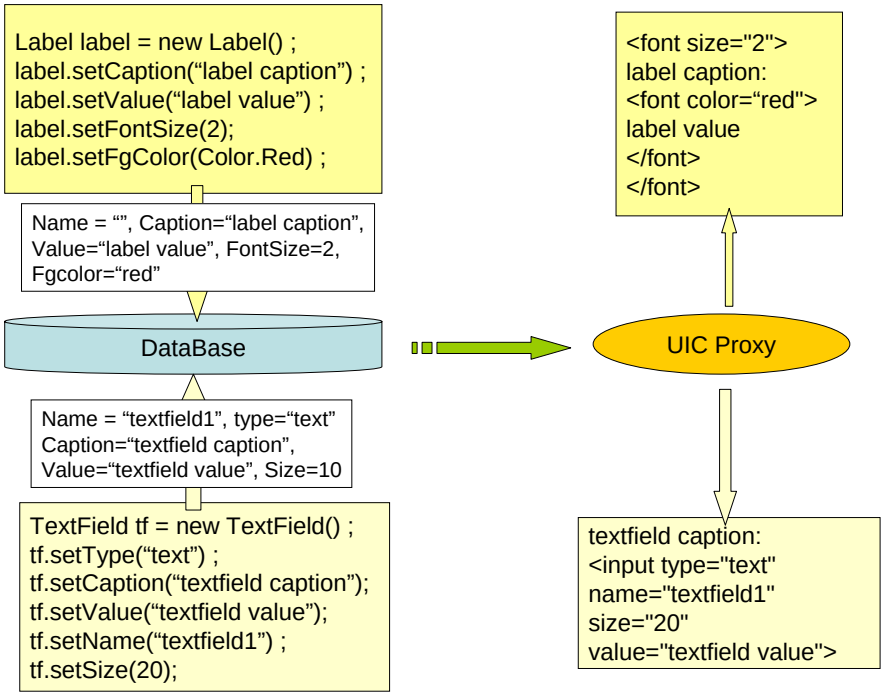


Figure 37: The translation example of Label and TextField.

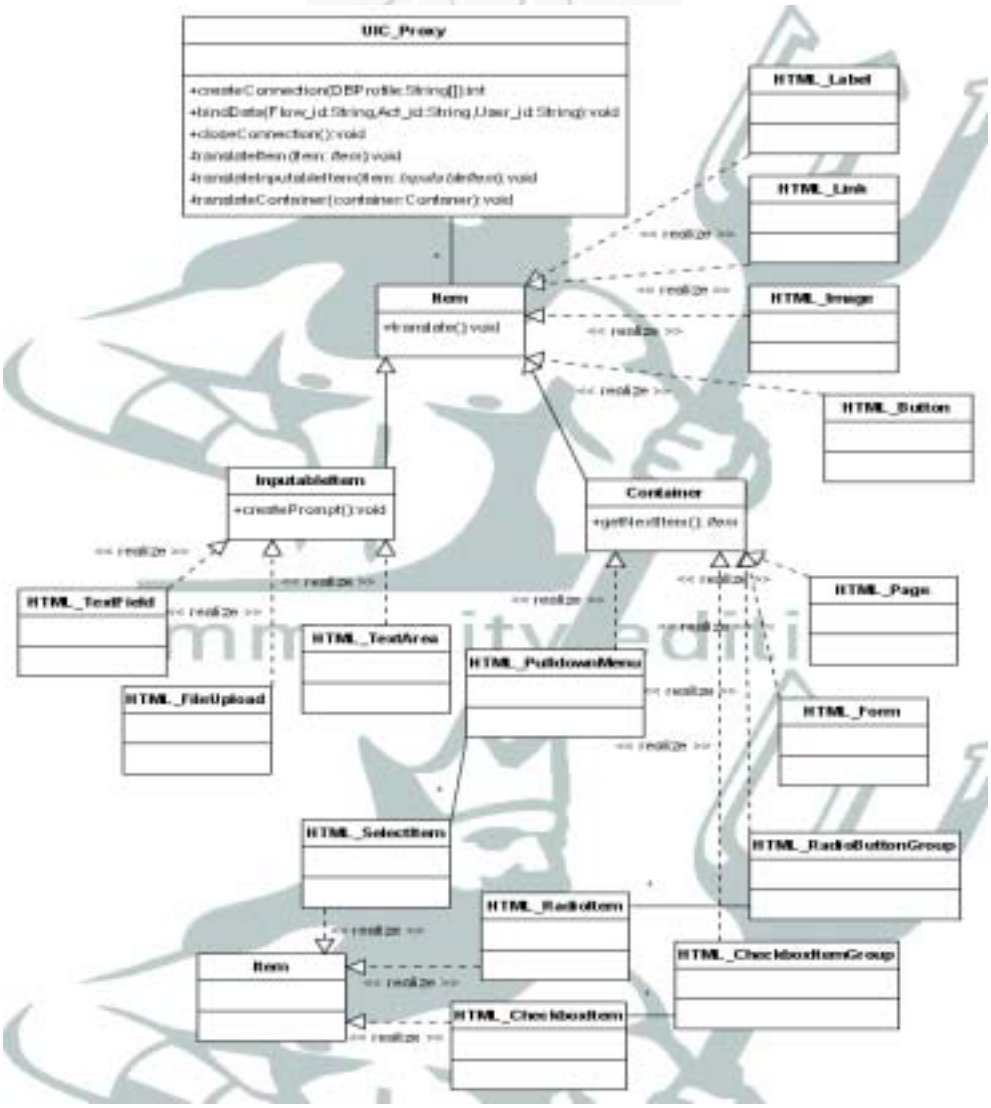


Figure 38: The class diagram of UIC_Proxy

Figure 38 is the class diagram of UIC_Proxy; the `UIC_Proxy` class manipulates other classes to translate data via interfaces. For instance, the `HTML_Label` class implements the Item interface and is responsible to translate the data of a *Label* into Html tags. By using interface, we can reduce the frequency of changing UIC_Proxy and change the output format by using different `UIC_Proxy`.

5.5.3.2 User Response Handling Mechanism

The user response handling includes two parts – the handling of asynchronous user response [8] and the data transformation between client and WfMS.

The key difference between a domain object and a human worker stems from the fact that while objects usually process messages synchronously, human workers handle work items asynchronously [8]. For example, WfMS will assign messages or forms to the users who participate in the execution of this activity but the users may not access these messages and make responses immediately. That is, the workflow participants may respond to requests asynchronously. However, the general-purpose programming languages like Smalltalk or Java implement message sends in a synchronous manner. In other words, upon sending a message, the sender waits until the receiver processes it and returns [18]. Some previous research solved this problem by returning a *Future objects* [8-18]. The usage of Future Objects is a technique from concurrent programming languages [2-3]. Future objects provide placeholders for objects whose identity is determined after the future objects representing them are created [3]. After sending message, a future object will be returned to the sender immediately, without waiting for the receiver to response. Once the real result becomes available, it replaces the future object transparently. However, the implementation of Future objects needs the programming language provide reflective capabilities, for instance, Smalltalk's `oneWayBecome:`.

In JOO-WfMS, we manipulate the operation of thread to handle asynchronous behavior. Activities in JOO-WfMS are executed in a thread. After storing the data of UICs into database (step 2 in Figure 34), developers can invoke the `waitResponse` method in the execution code of current activity. The code fragment of `waitResponse` method is listed in Figure 39:

```
public synchronized void waitResponse() {
```

```
while ("wait_user_response".equals( ActivityStatus )) {
try {
    ....
    //use the wait method of Object to pause the operatin of Activity
    wait();
} catch (InterruptedException e) {....}}}
```

Figure 39: The code fragment of "waitResponse" method of Activity object in JOO-WfMS

In `waitResponse` method we invoke the `wait` methd provided in `java.lang.Object` to pause the execution of an activity. The `wait` method causes current thread that owns this object's monitor to wait and releases the monitor of this object.The `java.lang.Object` is the parent of all Java objects so all Java objects inherit the `wait` method form `java.lang.Object`. Corresponding to the `wait` method, the notify method and notifyAll method in `java.lang.Object` can be invoked to weak up the waiting thread. Therefore, the `getResponse` method in `Activity` will be invoked by WfMS to resume the execution of the paused activity. In the `getResponse` method, we invoke notify mothod to weak up the paused thread. The code fragment of getResponse method is listed in Figure 40:

```
public synchronized void getResponse() {
    //use the notify method of Object to resume the operatin of Activity
    notify();
    ...
   }
```

Figure 40: The code fragment of "getResponse" method of Activity object in JOO-WfMS

To pause a running thread, we may also use the `sleep` method porvided in `java.lang.Thread`; the `sleep` method will cause the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds. To use the `sleep` method to pause the execution of a thread is simpler than using `wait`

and `notify` but the `sleep` mothod is not suitable for the user response handling in WfMS. In most cases, we can not predict when users will make response so we can not assign a specified number of milliseconds for the `sleep` method.

Next we will introduce the data transmission between client and WfMS. In most cases, the client programs (e.g. JSP) and server programs of a WfMS are executed in different processes in different machines. Therefore, how to transmit data between different processes is the problem we need to solve now.

Using database for the interchange of data can be a solution. The client programsand server programs interchange data by accessing the same database. However, we think that using database for the interchange of data is not suitable for the communication between client programs and WfMS because of the following reasons:

1. The using of database is more suitable for storing constant data. For instance, the message board or forum is more suitable to use database because the fields of them are not changed frequently. However, the user interfaces of different activities are different each other and the fields of them are also various, and therefore the design and maintenance of the tables will be another problem if using database for the interchange of data.
2. After the user makes responses, the waiting activity should be notified to weak up. The notifying mechanism will be complex and hard to implement if using database.

Therefore, we deliver data via TCP socket. A TCP socket will be established between client programs and WfMS and all the data will be delivered via this socket. Figure 41 illustrates the operation model of user response handling mechanism in JOO-WfMS:

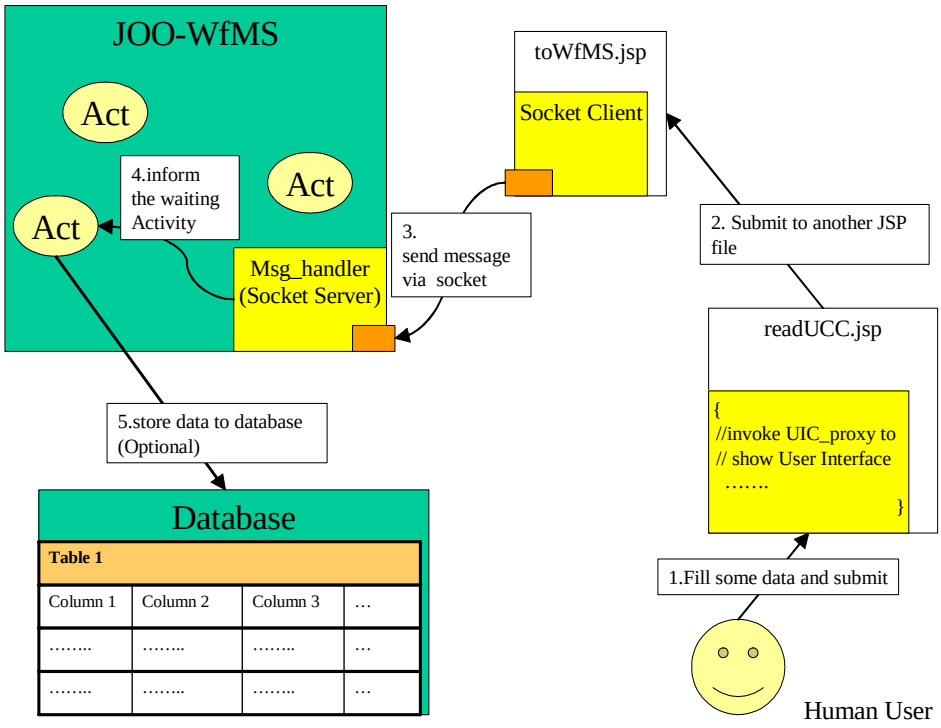


Figure 41: The operation model of User Response Handling Mechanism in JOO-WfMS.

Next, we introduce the steps of user response handling mechanism in JOO-WfMS below:

1. At first, the human user accesses the client program (readUCC.jsp) to fill form and then click the "submit button" to submit this form,
2. The form will then be delivered to another program (toWfMS.jsp). After receiving the form, this program will collect the data in this form and package them into a message. In addition to these data, the message may include other information, e.g. the id of an activity which is waiting for the message. After packaging these data, the program will establish a TCP socket with the server program (Msg_handler) of WfMS.
3. The program delivers message via the TCP socket built in step 2.
4. After receiving the message, the server program of WfMS will notify the waiting

activity according the information in the received message and set these data in this message into the waiting activity. The server program, Msg_handler, was introduced in section 5.3.1.

5. After executing, these data may be stored into database if needed.

In step 2, we package data into a message before establish a TCP socket. In addition to the data inputted by user, the message includes other information to identify the waiting activity and data for error detection. The data structure of the message is a Vector (`java.util.Vector`). The elements of the message are listed in Figure 42:

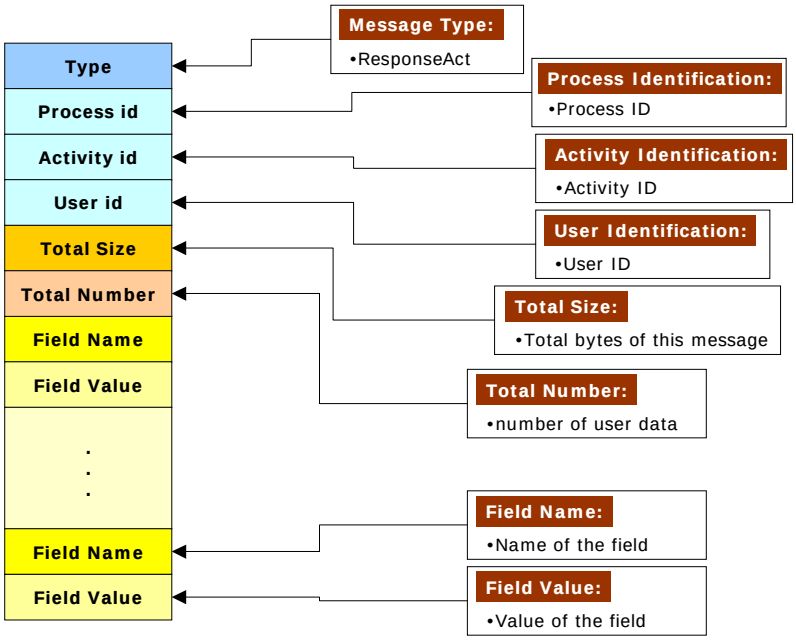


Figure 42: The structure of message content of user response.

1. **Type:** the first element is the type of this message. As mentioned at Section 5.4, message handler will receive message from users in three situations, therefore, the first element is used to identify the situation. The value is “ResponAct” in this situation.

2. **Process id:** the second element is the identification of process that the waiting activity belongs to.
3. **Activity id:** the third element is the identification of the waiting activity. After receiving message, the message handler will pass it to the specified activity according to the second and the third elements of this message.
4. **User id:** the fourth element is the identification of user who participates in the execution of waiting acitivity. Often, this identification is used for authorization.
5. **Total Size:** the value of fifth element means the size of this message. This value is used for error checking. The unit is Byte.
6. **Total number:** the sixth element is the number of elements of this message. This value is also used for error checking and to assist system in executing.

The six elements introduced before are assigned by program and the data inputted by users are kept in later elements. The values inputted by users are stored in a “Name, Value” pair. All the values are `String` type.

The second case that the message handler receives messages is the request for starting a process. Figure 43 illustrates the elements that comprise the message.

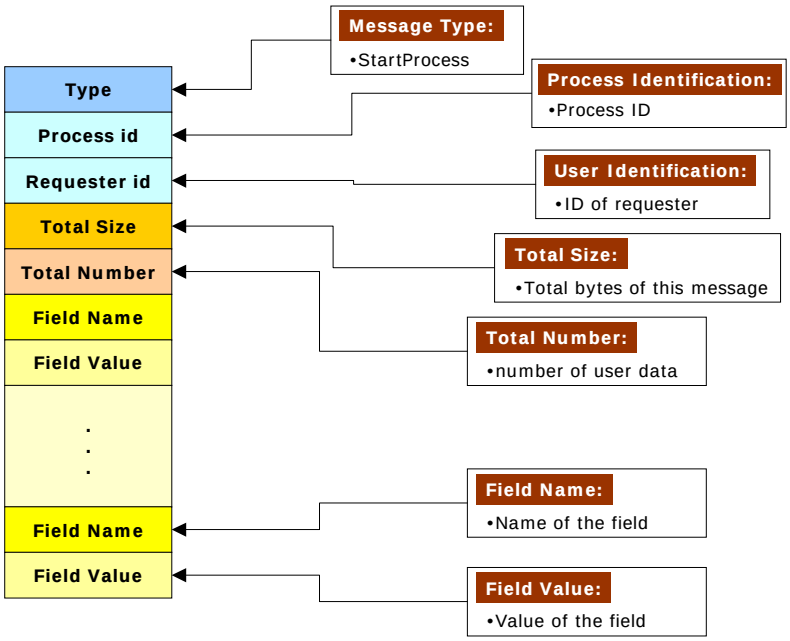


Figure 43: The structure of message content for starting a process

1. **Type:** the first element is the type of this message. The value is “StartProcess” in this situation.
2. **Process Def.** id: the second element is the identification of the specified process defintion.
3. **Requester id:** the third element is the identification of user who sends request to start a process with specific process definition.
4. **Total Size:** the value of fifth element means the size of this message. This value is used for error checking. The unit is Byte.
5. **Total number:** the sixth element is the number of elements of this message. This value is also used for error checking and to assist system in executing.
6. The data inputted by users are kept in the remaining elements of this message.

The final case of messages from users is the request for adding a new process definition into workflow system. The structure of this message is shown as following:

Type	Process Def. id	User id	Location of Process Definition file

Figure 44: The structure of message content for adding a new process

1. **Type:** the first element is the type of this message. The value is "AddProcessDef" in this situation.
2. **Process Def.** id: the second element is the identification of the specified process defintion. In this case, this identification may be assigned by process designers or generated by system.
3. **User id:** the third element is the identification of user who sends the request for adding a new process definition into system. This value should be used for authorization.
4. **Location:** the location of process definition file.

After recevinig messages, the message handler will execute some proper operations according to the message type.

In JOO-WfMS, we handle the asynchronous user response by manipulating thread and receive data from user by using TCP socket. We also design three message structures to aid the user data transmission. But there are still some problems to be overcome. Now, we only allow user to transmit the objects that have been defined in Java, the delivery of customized objects is not supported. In the future, we may support the customized objects by using reflection or Java RMI. And we also plan to use Java Message Service (JMS) to strengthen the asynchronous operation in JOO-WfMS.

5.5.4 Failure Recovery

As mentioned in section 4.3.2, the traditional failure recovery model specify the failure recovery specification in activities in a static way and therefore lacks of flexibility and reusability. In this research, we adopt WfFH to support a workflow failure-recovery model with the following features [13]:

1 The ECP and compensation set can be computed or derived during the workflow process run-time according to the execution results and status of workflow activities, as well as the type of failures.

2 The workflow designer can use the WfFH language to program the computation of the ECP and compensation. With the WfFM codes written by the workflow designer, the WfMS employs data-flow analysis technology to compute the ECP and compensation when the failure occurs.

3 The definition of failure recovery and compensation scope between activities can be absolutely independent. With this, inserting or deleting activities to or from a workflow process may not require modification of the failure-recovery specification in other activities. This reduces the maintenance overhead of workflow process dramatically and increases the reusability of workflow activities.

4 The details of information related to the failure can be sent to the ECP and activities in the compensation set to activate the most appropriate compensation for the failure situation that has occurred.

5 There can be multiple failures in a single activity, and each such failure can activate different failure-recovery processes.

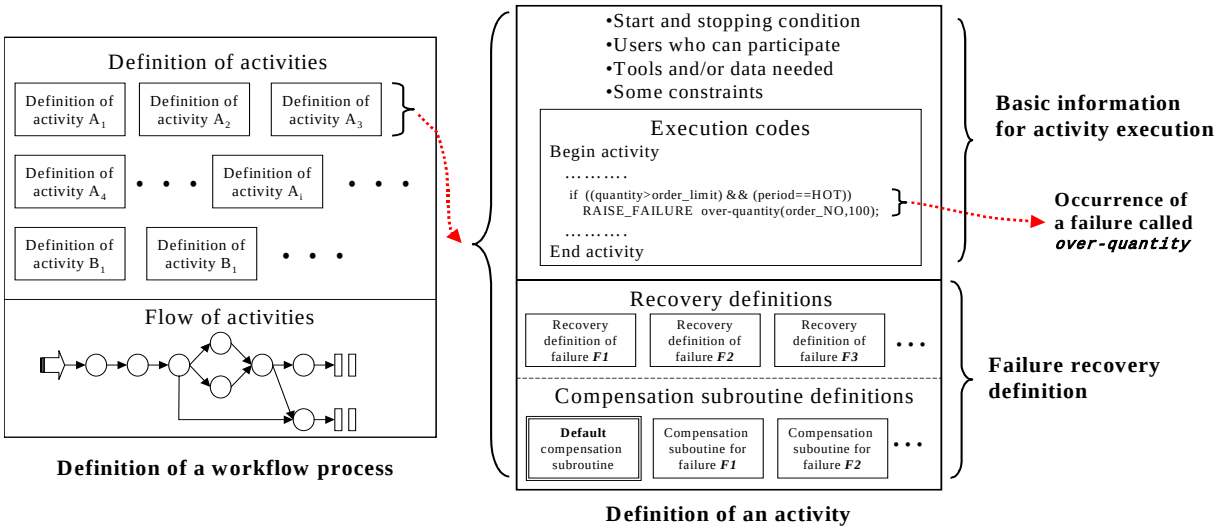


Figure 45: The definition of the workflow process and activities

The activity definition is shown in Figure 45 , which contains basic information for activity execution and additional failure-recovery definitions. The additional failure-recovery definition in the activity for data-flow-analysis-based failure recovery comprises *recovery definitions* and definitions of the *compensation subroutines*. The execution code may trigger the failure-recovery process when the execution causes a failure.

5.5.4.1 The Skeleton of a Recovery Definition

When a failure occurs, the workflow engine executes the WfFH codes of the corresponding recovery definition of the failure to compute the ECP and compensation set. All the activities in the compensation set should be compensated. In the compensation of an activity, the workflow engine first checks if there is a corresponding compensation subroutine of the occurred failure in the activity – this compensation subroutine will be executed if it exists; otherwise the *default* compensation subroutine will be executed. Since an activity may cause different failures, the activity definition may contain multiple recovery definitions and

compensation subroutines. According to the syntax of WfFH language, each recovery definition has the form shown in Figure 46.

```
   FAILURE Name(type1 arg1, type2 arg2, …)
1.1    BEGIN Declaration
1.2       // make declaration of variables
1.3       Activity_set a_set1, a_set2;
1.4       Resource R1, R2;
                …
1.5    END Declaration
2.1    BEGIN Compensation Set Computation
2.2      // code to derive compensation set here
                …
2.3      DO_Compensation a_set1;
2.4    END Compensation Set Computation
3.1    BEGIN ECP Computation
3.2      // code to make ECP computation set here
                …
3.3      Rollback_To a_set2;
3.4    END ECP Computation
   END Failure
```

Figure 46: The skeleton of a recovery definition written in the WfFH language

A recovery definition programmed with the WfFH language consists of a header and a body. The header begins with a keyword “FAILURE,” which is followed by the name of the failure and a sequence of arguments. The syntax of the argument list of the WfFH language is the same as that for Java [23]. These arguments have two uses: (1) to provide information for the computation of the ECP and compensation set, and (2) they will be passed to the compensation subroutines of the compensated activities. The compensation subroutines could use the received arguments to perform the most appropriate compensation. The body comes after the header, and comprises three sections. The first section in the body, the *declaration section*, includes declarations of variables used in the next two sections. Variables declared in the declaration section include:

- Variables used to store set of activities. The compensation set and ECP of a failure consist of set of activities. We can use the keyword “Activity_set” to make the declaration (see line 1.3 of Figure 46).

- Variables used to store resources. For data-flow-analysis-based specification and computation of the compensation set and ECP, we have to declare some variables which store shared resources accessed by different activities (see line 1.4 of Figure 46).
- Others. The WfFH language allows the use of all the data types defined in Java [23], including types defined by the user.

5.5.4.2 Methods in WfFH

The data-flow analysis technology (or global data-flow analysis technology) for compiler optimization analyzes how data values are modified across basic blocks of program statements [25]. The data-flow analysis technology used in our failure-recovery model is derived from the data-flow analysis technology used in compiler optimization. However, instead of analyzing the data values modified in program statements, our model analyzes the resource accesses of activities in workflow processes. To build up a data-flow-analysis-based failure-recovery mechanism, i.e., to compute the ECP and compensation set using data-flow analysis technology, the workflow engine which controls the execution of the activities has to monitor and record all the accesses to the *shared resources* of each activity. During the execution of a workflow activity, it may access private and/or shared resources. Private resources are defined as data used only within the execution of a single activity, whereas shared resources are data that are created, read, written, or modified by multiple activities, or data which are considered as the output results of the workflow process. In the current definition and implementation of the WfFH language, they are two kinds of resources that the activity can access: *files* and *database records*. The accesses to shared resources issued by activities are stored in an *execution-log database*. Each access record stored in the execution-log database

contains at least the activity which performs the access, the time and type of the access (i.e., insert, delete, update, or create), and the file name or the primary keys (or table name) of the access database records. File accesses are easy to monitor, but monitoring accesses to database records is much more difficult. We do this using execution-monitoring technology which is proposed and developed for reachability testing of client–server database applications [26, 27]. The next section in the definition is the *compensation-set computation* section, which the WfMS executes to derive the compensation set of this failure. Its syntax is similar to Java, with some extension of functions to support the computation of compensation sets. They can be classified into three main groups as described below.

The first group comprised functions which return the set of activities. We call them activity-manipulation functions:

- `ACTIVITY("name_of_activity")`. Given the name of an activity in the workflow process as the argument of this function (possibly containing the wildcard operator * or ?), this function returns the specified activities. For example, `a_set1=ACTIVITY("check*")` will return the set of activities whose names are prefixed with "*check*."
- `ACTIVITY(ALL)`. This function returns the set of all the activities in the workflow process.
- `ACTIVITY(EXECUTED)`. This function returns the set of all the activities which have been executed.
- `ACTIVITY(THIS)`. This function returns the set of the activity which causes the failure (the current activity).
- `ACTIVITY_READ(resource)`. This function returns the set of all the activities which performed a read operation on *resource*.

- ACTIVITY_WROTE(*resource*). This function returns the set of all the activities which performed a write operation on *resource*.
- ACTIVITY_READ_WROTE(*resource*). This function returns the set of all the activities which performed read and write operation on *resource*.
- ACTIVITY_DELETE(*resource*). This function returns the set of all the activities which performed the delete operation on *resource*.
- ACTIVITY_CREATE(*resource*). This function returns set of all the activities which created *resource*.
- FIRST(*activity_set*). This function returns the earliest executed activity in the *activity_set*.
- LAST(*activity_set*). This function returns the most-recent executed activity in the *activity_set*.

The second group comprises functions which return a set of resources. We call them resource-manipulation functions:

- SELECT_DB_RECORDS("*DB_Name*", "*select_sql_transaction*"). This returns the database records which are queries with the transaction *select_sql_transaction* from the database *DB_Name*.
- FILES(f_1, f_2, …, f_n). This returns the files f_1, f_2, …, f_n as a resource type.
- RESOURCE_READ_BY(*activity_set*). This returns all the resources read by *activity_set*.
- RESOURCE_WROTE_BY(*activity_set*). This returns all the resources written by *activity_set*.
- RESOURCE_DELETED_BY(*activity_set*). This returns all the resources deleted by *activity_set*.
- RESOURCE_CREATED_BY(*activity_set*). This returns all the resources created by *activity_set*.

The functions SELECT_DB_RECORDS and FILES return the specified resources from the shared resources of the workflow process. `RESOURCE_READ_BY`, `RESOURCE_WROTE_BY`, `RESOURCE_DELETED_BY`, and `RESOURCE_CREATED_BY` allow the user to determine what resources are accessed by activities.

The third group comprises the set of operations which support the activity- and resource-manipulation functions:

- `INTERSECT(S1, S2)`. This returns the intersection of activity sets or resource sets (i.e., `{x | x ∈ S1 and x ∈ S2}`).
- `UNION(S1, S2)`. This returns the union of activity sets or resource sets (i.e., `{x | x ∈ S1 or x ∈ S2}`).
- `SIZE_OF(S)`. This returns the number of elements in set S corresponding either to the set of activities or resources.
- `DIFFERENCE(S1, S2)`. This returns the set which deletes element of S_2 from S_1 (i.e., `{x | x ∈ S1 and x ∉ S2}`).
- `S(i)`. This returns the i-th element in set s.

The codes in this section also conform with the statement and loop structures of Java, which allows the recovery definitions programmed with the WfFH language to be easily translated into Java objects. A statement specifying the resulting compensation set should appear at the end of this section, consisting of a keyword "`DO_Compensation`" and an activity set variable which stores the resulting compensation set (see line 2.3 of Figure 46).

The final section in the recovery definition for the WfFH language is the *ECP*

computation section. The programmer uses this to specify how to compute the ECP. The syntax and available functions of this section are identical to those for the compensation-set computation section described above. Furthermore, it inherits the computation results of all the variables used in the section in which the compensation set is computed. The ECP computation section also ends with a statement to specify the resulting ECP. It begins with a keyword "Rollback_To" which follows an activity-set variable which stores the resulting ECP.

5.5.4.3 How JOO-WfMS supprot WfFH

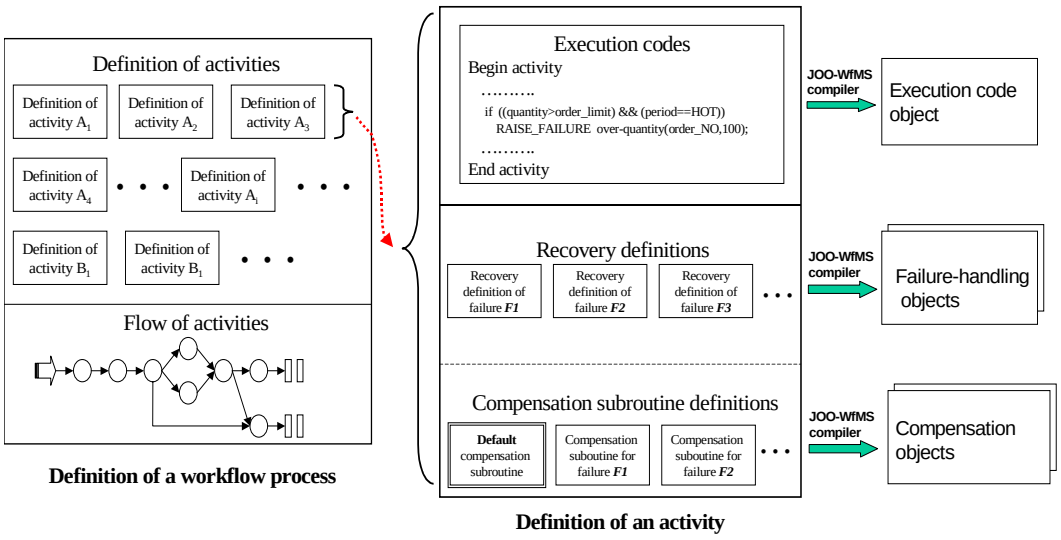


Figure 47: Generating Java object codes of a workflow process

In JOO-WfMS, we use code generation technology to translate WfFH to Java codes. Figure 47 shows the workflow process definition which conforms to the syntax of the JOO-WfMS as compiled using the *JOO-WfMS compiler* to generate Java object codes (or Java class files), including some activity objects, a flow object, some failure-handling objects, and some compensation objects. An activity object includes information and codes for executing the activity. Also, each activity object is

associated with some failure-handling objects and compensation objects. The flow object is used to control the order of execution of the activities. These Java objects are sent to the JOO workflow engine which activates and controls the execution of the workflow process encoding by these Java objects. There are standard Java interface definitions for workflow, flow control, activity, execution code, failure, and compensation handler objects in JOO-WfMS.

The way to activate the failure-recovery process is specified in the execution code of the definition of the activity. In the JOO-WfMS, the execution code of an activity is also a Java program with the *failure-recovery extension*, which is of the following form:

```
RAISE_FAILURE Failure_Name(arg1, arg2, …,argn);
```

The JOO-WfMS compiler translates the above statement into the following Java program:

```
//To generate an array args to store arg1, arg2, …, and argn
 Object[] args= new Object[n];
 args[0]= arg1;
 args[1]=arg2 ;
    …
 args[n]=argn ;
 // To instantiate a failure object
 Failure fail = activity.getFailure(Failure_Name) ;
 //To  setup the failure arguments
 fail.SetArguments(args) ;
 // To throw a Java exception to start the failure recovery process
 throw new RaiseFailureException(fail) ;
```

The following is a fragment from the execution code of the activity definition of A_6. It specifies how to activate recovery process of the *inventory_insufficient* and *over-quantity* failures:

```
…
if (inventory < ordered_quantity)
     RAISE_FAILURE inventory-insufficient( );
if ((quantity > order_limit) && (period==HOT))
     RAISE_FAILURE over-quantity(order_No, order_limit);
…
```

The "inventory", "ordered_quantity", "quantity", "order_limit", "period" and "order_No" are local variables in the execution codes of A_6. If the value of "inventory" is less than "ordered_quantity", it will activate the failure recovery according to the recovery definition named "*inventory-insufficient*". Also, if "quantity" is greater than "order_limit" and the "period" is "HOT," the recovery for an over-quantity failure will start. The code translated by JOO-WfMS compiler is shown as follows:

```
if (inventory < ordered_quantity) {
  Failure fail =activity.getFailure("inventory-insufficient") ;
  throw new RaiseFailureException(fail) ;
}
if ((quantity > order_limit) && (period==HOT)) {
  Object[] args= new Object[2] ;
  args[0] = order_No ;
  args[1] = order_limit ;
  Failure fail =activity.getFailure("over-quantity") ;
  fail.SetArguments(args) ;
  throw new RaiseFailureException(fail) ;
}
```

Figure 48 shows the WfFH code of "Over-Quantity" failure and the Java Code after translated by JOO-WfMS compiler.

Over-Quantity failure in WfFH	Java Code

FAILURE over-quantity (int order_No, int order_Limit)
 	
//---------Variable declarations----------
BEGIN Declaration
Activity_Set a_set1;
Activity_Set comp_set, ecp;
Resource Oinventory, OQuantity;
int i, size;
END DECLARATION
	public class Failure_Over_Quantity extends Failure
	{
	//---------Variable declarations----------
	ActivitySet a_set1 ;
	ActivitySet comp_set, ecp ;
	ResourceSet Oinventory, OQuantity;
	int i,size;
	
	//----------Methods to hanlde arguments of failure-----
	public void SetArguments(Object[] args){
	for(int i = 0 ; i < args.length ; i++)
	failureArgs.addElement(args[i]);
	}
	public Object[] GetArguments(){
	Object[] obj = new Object[failureArgs.size()];
	failureArgs.copyInto(obj);
	return obj;
	}
	public Class[] GetArgumentTypes(){
	Class[] cla = new Class[failureArgs.size()];
	for(int i = 0 ; i < failureArgs.size() ; i++)
	cla[i] = failureArgs.elementAt(i).getClass();
	return cla;
	}

```
//-----------------Compute compensation set -----------
  BEGIN Compensation Set Computation
    OQuantity=SELECT_DBRECORDS("WF_DB1", "SELECT quantity
      FROM OrderForm WHERE OrderForm.OrderNo=" + Order_No);
    a_set1 = ACTIVITY(EXECUTED);
    size = SIZE_OF(a_set1);
    for (i = 1; i <= size ; i++)
    {
      if          (INTERSECT(RESOURCE_WROTE_BY(a_set1(i),
      OQuantity) != EmptySet)
        comp_set = UNION(comp_set, a_set1(i));
    }
    DO_Compensation comp_set;
  END Compensation Set Computation




//-------------Compute  ECP -------------------------
  BEGIN ECP Compution
    ecp = FIRST(comp_set);
    Rollback_To ecp;
  END ECP Compution
END FAILURE
```

```
//-----------------Compute compensation set ----------
  public  ActivitySet CompensationSetComputation() {
      comp_set = new ActivitySet () ;
      int                    Order_No                    =
((Integer)GetArguments()[0]).intValue();
      OQuantity=SELECT_DBRECORDS("WF_DB1",
         "SELECT    quantity    FROM    OrderForm    WHERE
         OrderForm.OrderNo=" + Order_No);
      a_set1=ACTIVITY_EXECUTED(
       activity.GetBelongWorkflow().GetWorkflowID());
      size=SIZE_OF(a_set1) ;
     for (i=1,i<=size;i++)
     {
      if          (INTERSECT(RESOURCE_WROTE_BY(a_set1(i),
      OQuantity) != EmptySet)
         comp_set = UNION(comp_set, a_set1(i));
     }
      return comp_set ;
  }
//-------------Compute  ECP -------------------------
  public ECP ECPComputation() {
      ecp=FIRST( comp_set ) );
      return new ECP(ecp);
  }
}
```

Figure 48. Translating of the WfFH code into the Java program

JOO-WfMS offers a flexible and reusable failure recovery mechanism via a flexiable failure recovery model and WfFH. However, there are some issues for WfFH. An important issue is how we use WfFH in a transactional environment. Can we implement the ACID (Atomicity, Consistency, Isolation and Durability) properties of transaction execution bu using WfFH? Another important issue is to use the WfFH in a distributed scenario. Failure recovery involve rolling back to previous activities, however, to navigate the workflow processes between different WfMSs is more complex than that in a single WfMS.

5.6 Workflow Transition

As mentioned before, flow control of process is essential for workflow system but the object-oriented deempasize the representation of flow control. Therefor, we must provide abstraction about flow control when designing architecture of WfMS. Currently, most workow languages support the basic constructs of sequence, iteration, splits (AND and OR) and joins (AND and OR) [33]. Usually, the execution of process

(or brach of process) will wait when meeting with“Join”structures until the “join condition” is met, and the system will decide the next activities that will be performed according to some execution results when meeting “Split”structure. The semantics of these control structures extend those of Petri nets but the interpretation of these basic constructs is not uniform and it is often unclear how more complex requirements could be supported. Refer to [20][33][34][35], we try to define and extend these structures as following:

Sequence

We refer to the sequential performance of activities when these have to be carried out one after another [35]. It two activities need to be carried out sequentially, there usually is a clear interdependence between them. For example, the result of the first is required in order to perform the second. The sequence structure is the most common in workflow and Figure 49 illustrates this structure:

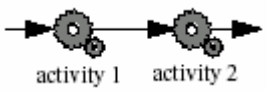


Figure 49: The sequence structure [20]

OR-Join

In OR-Join structure, the successors may be performed as soon as one of the predecessors has been finished. That is, in OR-Join structure, the “join codition” will be met as soon as one of the predecessors has been completed. In this paper, we represent an OR-Join by using the symbol “∨”on the output side, Figure 50 illustrates an example of OR-Join structure.

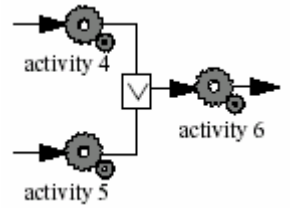


Figure 50: The OR-Join structure [20]

AND-Join

In this structure, the successors cannot be performed until all predecessor activities have been finished. That is, the “join codition”will be met when all the predecessor activities have been completed. We represent the AND-Join by using the symbol“∧”on the output side. An Example of this structure is shown in Figure 51.

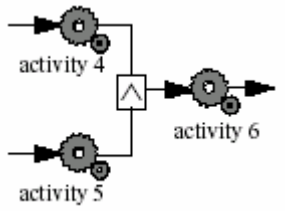


Figure 51: The AND-Join structure [20]

SELECTIVE-Join

In this structure, the “join codition”will be met when the specified successors have been completed. The SELECTIVE-Join is a complex join structure, but it can be implemented by putting AND-Join and OR-Join structures together. We represent the SELECTIVE-Join by using the symbol“X” on the output side, an example of the structure is shown in Figure 52.

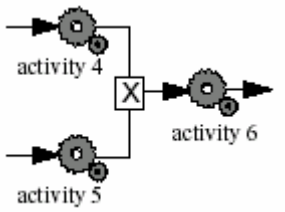


Figure 52: The SELECTIVE-Join structure [20]

OR-Split

OR-Split, i.e., only one of the specified successors may be triggered by the predecessor activities. We represent the OR-Join by using the symbol “∨” on the input side. Figure 53 illustrates an example of OR-Split structure.

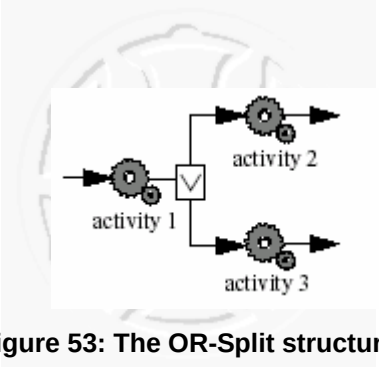


Figure 53: The OR-Split structure [20]

AND-Split

All the successors will be triggered to execute when meeting with AND-Split structure. We represent the AND-Join by using the symbol "^" on the input side. Figure 54 illustrates an example of AND-Split structure.

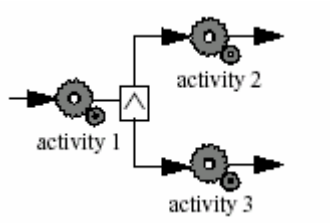


Figure 54: The AND-Split structure [20]

SELECTIVE-Split

In SELECTIVE-Split structure, only a set of the specified successors will be chosen to execute. The SELECTIVE-Split is a complex split structure, also, it can be implemented by putting AND-Split and OR-Split structures together. In fact, OR-Split is a simplified SELECTIVE-Split. We represent the SELECTIVE-Split by using the symbol"X" on the input side, as shown in Figure 55.

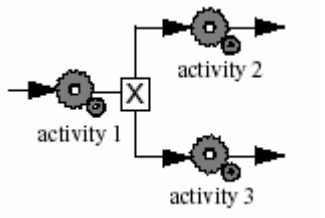


Figure 55: The SELECTIVE-Split structure [20]

Every WfMS has its way to implement or support these flow control structures, and so does JOO-WfMS. *Transition* component is our solution for flow control in

JOO-WfMS. Figure 56 illustates the structure of transition component. Transition includes two objects – *Entry* and *Exit*, system developer should implement the“join condition”in the onEnter method of Entry, and implement the algorithm for deciding the activities that will be performed next in onExit method of Exit. With different Entry and Exit, transition can deal with different Join and Split condition.

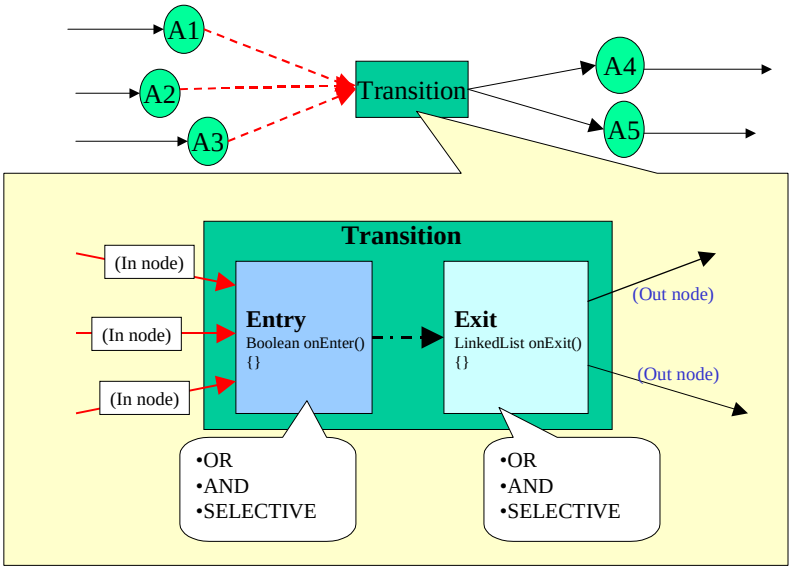


Figure 56: the structure of Transition

In JOO-WfMS, we provide several Entry and Exit objects to support the basic and complex flow control structures. We list these objects and the code fragements in Table 3.

Transition	`public class Transition {`
	`  Entry entry = new Entry () ;`
	`  Exit exit = new Exit() ;`
	`  Vector resultPath = new Vector () ;`
	`  Process belongProcess = null ;`
	`  …`
	`  public synchronized void decide() {`
	`  if (entry.onEnter()) {`
	`  //if”join condition”is met`
	`  nextActs = exit.onExit() ;`
	`  //decide the next Activities`

	… } } … }

Entry	public class Entry {
	Vector in_node = new Vector() ;
	…
	public boolean onEnter() {
	//default: return true
	return true ;
	}
	…
	}

OR-Entry	public class OrEntry extends Entry{
	String Acomplete = ActivityConst.Complete ;
	String Tcomplete = TransitionConst.Complete ;
	boolean pass = false ; int i = 0 ;
	public boolean onEnter() {
	while (i< in_node.size() && !pass) {
	Node n=(Node)in_node.get(i) ;
	String state = n.getState() ;
	//pass will be true if any predecessor is finished.
	if(Acomplete.equals(state)||
	Tcomplete.equals(state))
	pass=true;
	i++ ;
	}
	return pass ;
	}
	…
	}

AND-Entry	public class OrEntry extends Entry{
	String Acomplete = ActivityConst.Complete ;
	String Tcomplete = TransitionConst.Complete ;
	boolean pass = true ;
	public boolean onEnter() {
	for (int i = 0 ; i< in_node.size() ; i ++) {
	Node n=(Node)in_node.get(i) ;
	String state = n.getState() ;
	//pass will be false if any predecessor is not
	//finished.
	if(!Acomplete.equals(state)||
	!Tcomplete.equals(state))
	pass=false;
	}
	return pass ;
	}
	…
	}

Exit	public class Exit {
	Vector out_node = new Vector() ;
	Vector nextActs = new Vector() ;
	…
	public Vector onExit() {
	//default: return the first successor
	nextActs.add(out_node.get(0)) ;
	return nextActs ;
	}
	…
	}

AND-Exit	public class ANDExit extends Exit {
	  …
	  public Vector onExit() {
	  //return all successor
	  for (int i = 0 ; i < out_node.size() ; i++)
	  nextActs.add(out_node.get(i)) ;
	  return nextActs;
	  }
	}

Table 3: List of Entry and Exit objects provided by JOO-WfMS

Developers may assign different Entry and Exit objects to transition to implement different flow control structures, i.e. transition with AND-Entry object and Exit object can be regarded as AND-Join, transition with OR-Entry and Exit can be viewed as OR-Join, and Entry operates in coordination to AND-Exit can be regarded as AND-Split.

We do not provide SELECTIVE-Entry/Exit and OR-Exit objects in JOO-WfMS, because the “join condition”and algorithm for choosing next activities of these objects should be implemented by workflow component developers or flow designers specifically. We provide some supports for developers to customize the flow control structures. Developers can get the information about incoming activities (e.g. the state of activity) from in_node of Entry and the information about outgoing activities from `out_node` of Exit. `in_node` and `out_node` are “Vector”s and comprise Nodes used in flowchart. `belongProcess`, the member variable of Transition, is the reference of the process instance that this transition belongs to, developers may use this to access variables and resources kept in the process instance. Moreover, the data-flow analysis technology used in WfFH [13] can also be used to access activities and resources in the workflow.

5.6.1 Design Details

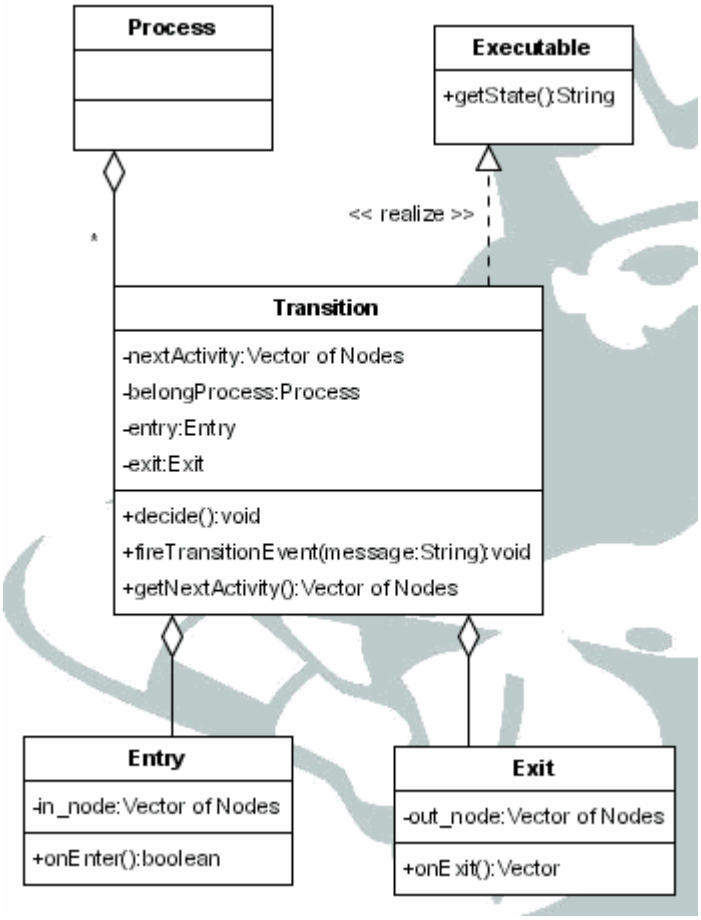


Figure 57: the class diagram of Transition

Figure 57 is the simplified class diagram of Transition. When receiving complete event from successors (e.g. after the activity A1 in Figure 56 is finished), Process instance instantiates the Transition (this action will not be performed if the Transition instance has existed). After instantiating Transition, the Process will instantiates Entry and Exit objects according to the information kept in flowchart, and in_node and out_node are assigned to Entry and Exit in this moment. As shown in Table 3, the `decide` method will be invoked to decide the next activities. The execution results will be stored in `nextActivity` and can be retrieved by invoking `getNextActivity` method.

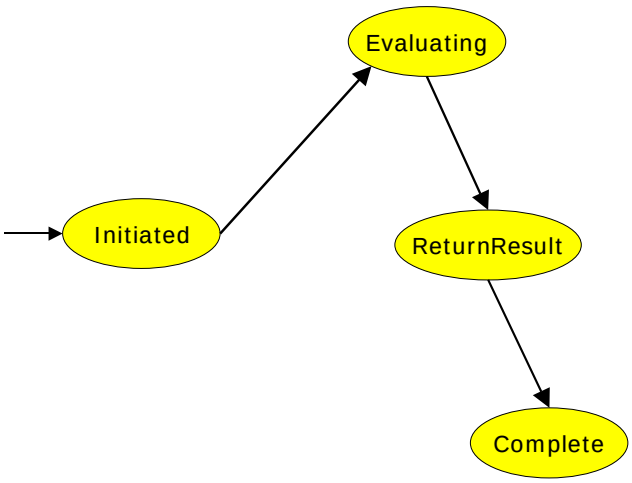


Figure 58: the state transition for transition instance in JOO-WfMS

The state transition of transition instance in JOO-WfMS is shown in **Figure 58**, we explains these states below:

- *Initiated*: the initial state of transition instance is “initiated”, this state reveals that the transition object has been instantiated but the decide method has not yet been invoked.
- *Evaluating*: the decide method has been invoked but the results have not been returned (e.g. the”join codition”is not met)
- *ReturnResult*: the execution of decide method has completed and the selected successors have been returned.
- *Complete*: execution of the transition instance has completed.

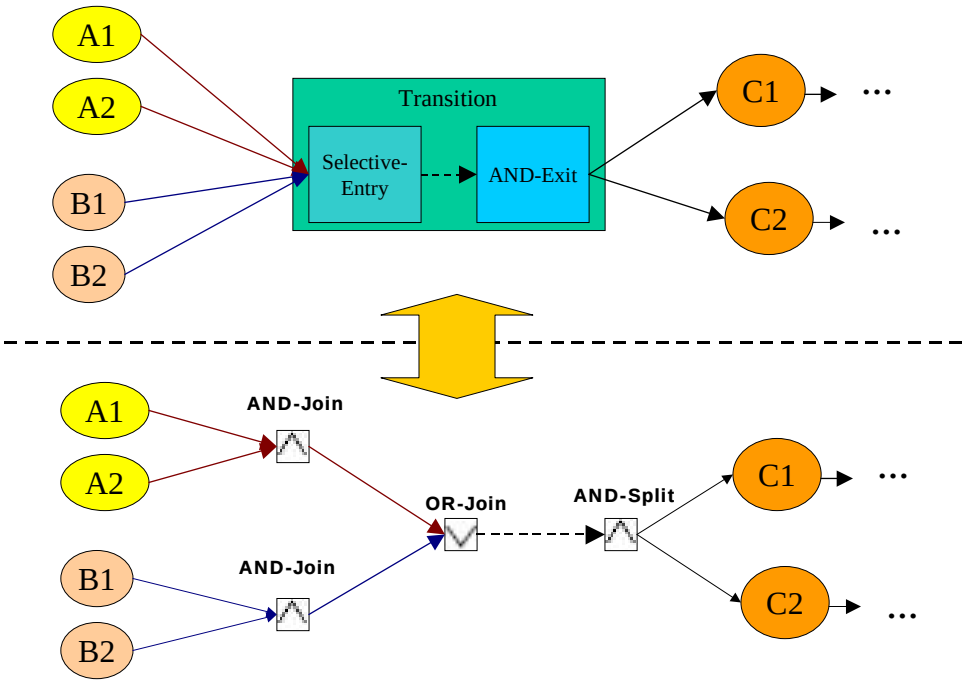
Different to Process and Activity, the state transition of Transition instance is automatic and cannot be controlled by mothod invocation.

5.6.2 Discussion of Workflow Transition

The transition component in JOO-WfMS comprises Transition object, Entry object and Exit object. Because the “join condition” and the navigation decision algorithm

are implemented within Entry and Exit, the change of condition and algorithm does not influence Transition object. In general, the Transition object can be fully reused. Moreover, developers may use the Entry objects and Exit objects provided in JOO-WfMS to implement basic flow control structures.

The selective structures, e.g. SELECTIVE-Entry and SELECTIVE-Exit, should be implemented by workflow component developers or flow designers specifically. However, these objects can be more reusable and more flexible by adopting data-flow analysis technology used in WfFH [13]. An example is shown in the following figure; the"join condition"of Selective-entry is"both activity A1 and A2 must be complete"or"both activity B1 and B2 must be complete".



The"join condition"may be coded as following:

```
Node a1 = (Node)in_node.get(0); Node a2 = (Node)in_node.get(1);
Node b1 = (Node)in_node.get(2); Node b2 = (Node)in_node.get(3);
```

```
String complete = ActivityConst.Complete ;
if (complete.equals(a1.getState())&&complete.equals(a2.getState())||
   complete.equals(b1.getState())&&complete.equals(b2.getState()))
…
```

Figure 59: Code example for join condition of transition in JOO-WfMS (1)

These codes may be modified when the process changes, e.g. delete activity B1 or add activity A3 into process and change the“join codition”to be“activity a1, a2 and a3 should be all complete”and“both activity B1 and B2 must be complete”. By using the activity-manipulation and operator functions provided in WfFH, the“join condition”can be coded as following:

```
…
Set a1 = ACTIVITY(“A*”) ;  Set b1 = ACTIVITY (“B*”) ;
Set executed = ACTIVITY (EXECUTED) ;
boolean pass1 = true, pass2 = true ;
if (SIZE_OF(INTERSECT(a1, executed)) != SIZE_OF(a1)) pass1 = false ;
if (SIZE_OF(INTERSECT(b1, executed)) != SIZE_OF(b1)) pass2 = false ;
boolean pass = pass1 || pass2 ;
   …
```

Figure 60: Code example for join condition of transition in JOO-WfMS (2)

These codes don’t need to be modified when changing the process. Because we use activity-manipulation function (`ACTIVITY(“Name”)` and `ACTIVITY(EXECUTED)`) to execute the activity whose names are prefixed with “A”or“B” and the activities that have been complete during the run-time of process, instead of specify these activities in build-time. Therefore, we think our design can support the reusability to Transition components, and the reusability we defined for Transition can be reached by using Transition components together with the activity-manipulation and operator functions used in WfFH.

5.7 Workflow Participants

It is common for WfMS to involve users in the execution of workflow processes or activities. Therefore, most workflow systems have their own model (we call it as

organizational model in this paper) to represent the workflow users and manage the information about them. In JOO-WfMS, we offer three basic objects for workflow participants – Human, Organization and Role. We list the descriptions and attributes of these three objects in Table 4.

Human represents a single user, e.g. an employee, a manager or a student
● **id**: the unique identification of the user; in current implementaion, the id of a human is the string that is prefixed with " H ", e.g. " H0001 ". ● **name**: name of the user. ● **description**: Short textual description of the user. ● **belongs:** the groups that the user belongs to. ● **roles**: the roles that are played by the user.
Organization represents a group, e.g. a department, a company or a college.
● **id**: the unique identification of the group; in current implementaion, the id of a organization is the string that is prefixed with " O ", e.g. " O0001 ". ● **name**: name of the group. ● **description**: Short textual description of the group. ● **belongs:** the groups that this group is subordinate to. ● **managers:** the managers of this group. ● **members:** users who belong to this group.
Role represents a function a human has within a organization, e.g. principal, manager
● **id**: the unique identification of the role; usually, we use the string prefixed with " R " for id, e.g. " R0001 ". ● **name**: name of the role. ● **description**: Short textual description of the role. ● **belongs:** the groups that this role belongs to. ● **actualusers:** users who paly the role.

Table 4: The basic objects of participant in JOO-WfMS

According to the table, a user can play one or more roles and many users may play the same role. In a group, there may be one or more leaders and more than one member, a user may belong to more than one groups. Every user, group and role may have his identification and the identification may be unique. In JOO-WfMS, we use this unique identification as primary key to retrieve information from database.

When implementing processes and activities that involve users to execute, we meet a problem – the relative relation between users. Usually, we assign user in a absolute way when designing workflow process. For example, we may assign a role whose id is”R0005”to participate the execution of activity A_n. Implementation of organizational model of most WfMS support this way [20][43][44]. However, in some cases, we cannot accurately know that which users will participate in. For instance, we want to design a process for students in an university to ask for leave, the student who want to ask for leave should fill a form first and then the form should be sent to his teacher for instruction. But, when designing process, we cannot predict who will ask for leave, therefore, we cannot assign the teacher for instruction accurately. User who is responsible for writing instruction on the form is”the teacher of the student who asks for leave”. “The teacher of a student”is a relative expression, that is, we may assign this user in a relative way. To support the use of relative relation, we try to use the tree structure to represent the organizational model. The leaf of the tree can be human, organization or role. Figure 61 is an example of organizational model in JOO-WfMS. The graph in Figure 61 illustrates a part of teaching departments and administration departments of a university and the relationship between them. We use rectangle to represent the organization and circular to represent human or role.

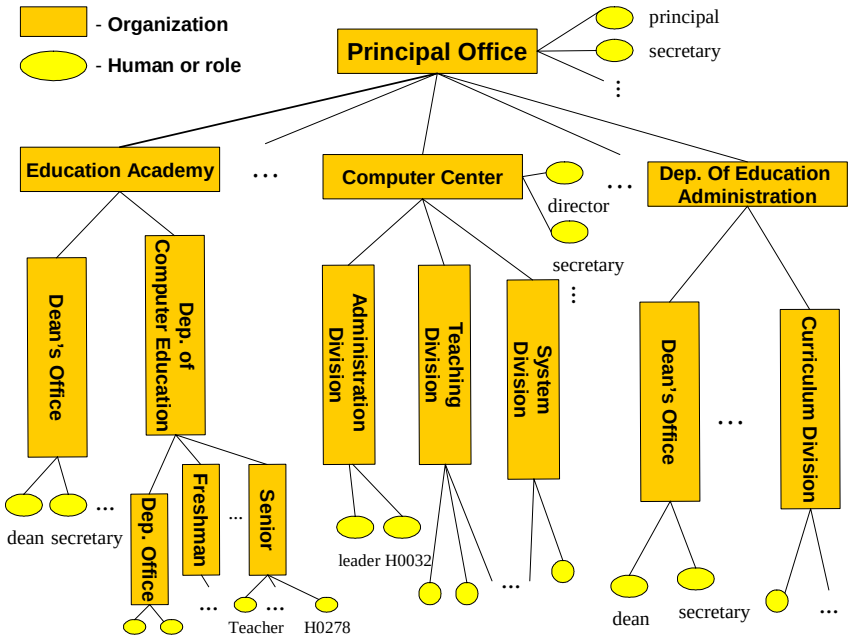


Figure 61: an example of organizational model in JOO-WfMS

By tracing the tree structure, the relative user can be found during the run-time of process. For the example mentioned before, a student (e.g. H0278) of computer education department wants to ask for leave, we can find the organization (Senior) that the student belongs to (via the `belongs` attribute of human) and then find the teacher (via the managers attribute or other extended attributes of organization). Furthermore, we can find the other users (e.g. the principal or the chief of computer center) by tracing the tree structure.

5.7.1 Design Details

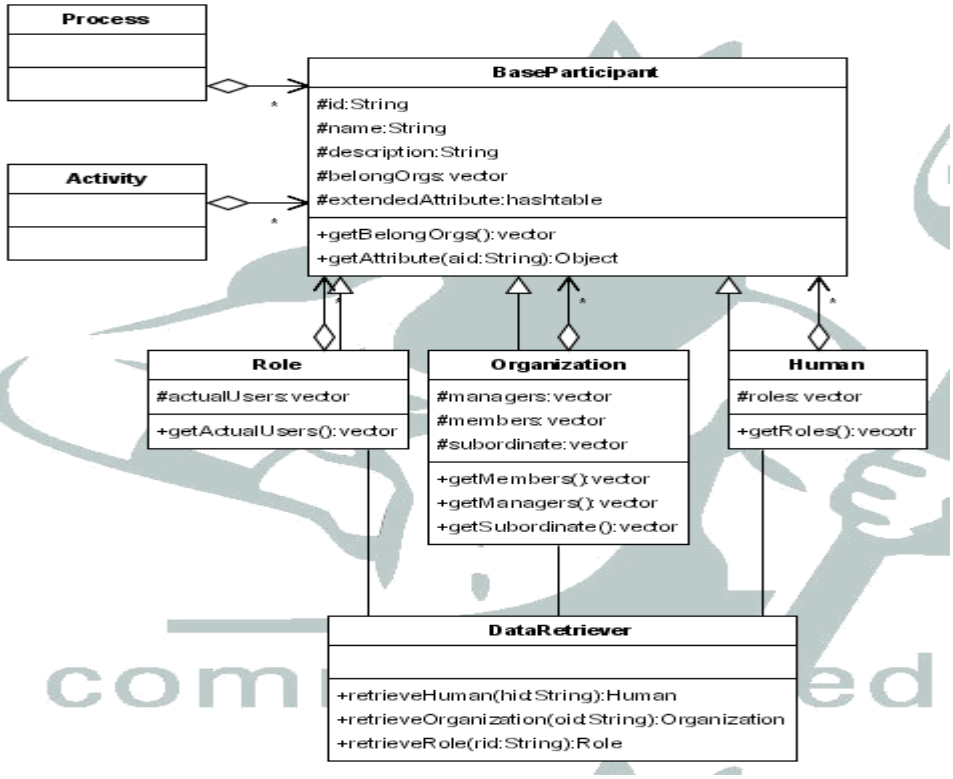


Figure 62: the class diagram of participant in JOO-WfMS

Figure 62 is the class diagram of workflow paticipant in JOO-WfMS. `Human`, `Organization` and `Role` inherit the `BaseParticipant` class. To support the tree structure, we add the `subordinate` attribute to `Organization` to keep the information about the subordinate organizations. The `getBelongOrgs` and `getSubordinate` methods can be used to traverse the tree structure. The `getBelongOrgs` method will return the references point to the predecessor organization instances, and the getSubordinate method will return the references point to the successor organization instances. When designing process, the process designer may only use some brief informantion (e.g. id) to assign the participant to process or activity; the details about the participant are recorded in database. Therefore, the `DataRetriever` is used to retieve the data of `Human` and `Organization` and `Role` instance from database (or other data storage) during the run-time of process. The `retrieveHuman` method will query the data from databse according to the id

of a human and return a `Human` instance. Similarly, the other two methods, `retrieveOrganization` and `retrieveRole`, will reture an `Organization` instance and a `Role` instance.

5.7.2 Discussion of Workflow Participants

In JOO-WfMS, we offer some classes for the implementation of workflow participants. We separate the database manipulation functions from `Human`, `Organization` and `Role`; we believe this design can promote the reusability of these classes. We can change the DAO used by `DataRetriever` or modify the `DataRetriever` class to meet the changes of data storage, instead of modifying `Human`, `Organization` and `Role` classes. The `BaseParticipant` class offers a member variable, `extendedAttribute`, for programmers to add new attributes. Programmers can extend `Human`, `Organization` and `Role` by using `extendedAttribute` rather than inheriting them.

For the problem of relative relation between users, we use the tree structure to deal with it. But another problem is that most workflow model languages lack the support for this kind of expression, including XPDL [11]. The workflow process in JOO-WfMS is represented with XPDL; however, XPDL does not support the expression of relative relation in current version. The code fragment shown below is an example of a particpant declaration in XPDL [11]:

```
<Participants>
      <Participant Id="R0002">
         <ParticipantType Type="ROLE"/>
         <Description>Order shipper</Description>
      </Participant>
</Participants>
```

Figure 63: Code example of the description of participant in XPDL

The syntax is more suitable for the expression of absolute relationship. In the future, we may try to extend the expression of XPDL or define new syntax to express the relationship between users.

5.8 Persistence

Although the Persistence functionality is not essential for the execution of workflow management system, most workflow systems implement functions or components for it. As mentioned before, workflow system may keep some information for evaluation and recovery purpose. In this paper, we call the information kept by Workflow management system “history”or”execution log”. Workflow management systems may use database management systems (DBMS) to store the history. Most commercial products use relational database management systems (RDBMS) to store history. Some techniques (e.g. the Data Access Object pattern introduced in J2EE) and standards for database connectivity (e.g. ODBC and JDBC) help us to manipulate database systems. In fact, in addition to using database, using text files to keep the execution logs is also common. Therefore, the persistence component should allow developers to use different data storages.

5.8.1 Design Details

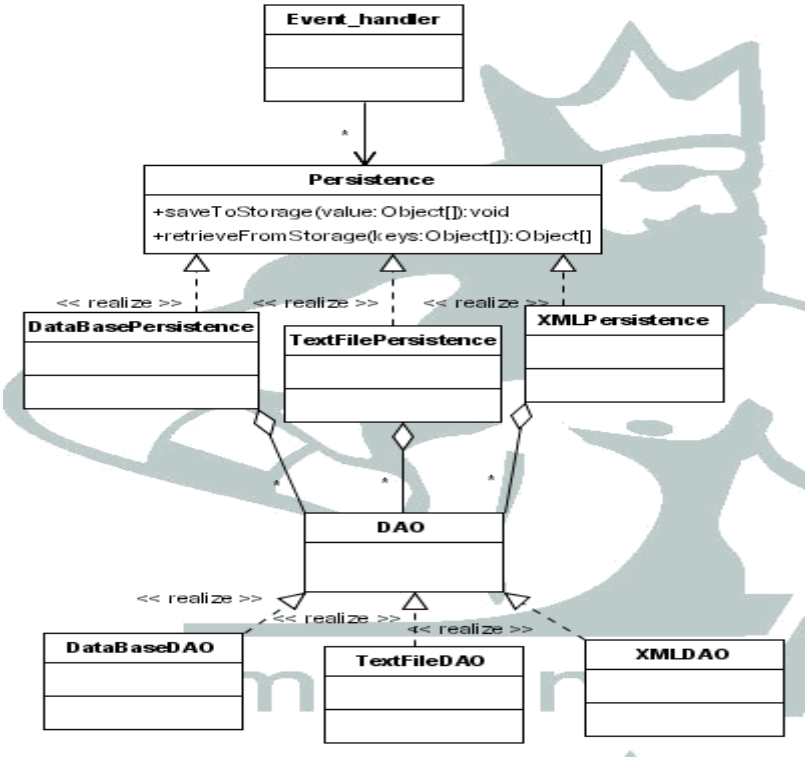


Figure 64: the class diagram of Persistence

Figure 64 is the class diagram of Persistence component. The event handlers of workflow engine or process will invoke the implementations of the `Persistence` interface (e.g. the DataBasePersistence, TextFilePersistence or XMLPersistence) to store the logs into database, text files or XML format. There are two methods defined in `Persistence` interface – `saveToStorage` method and `retrieveFromStorage` method. Developers may implement the policy about saving history (e.g. which data should be kept or the history should be stored into which table in database) in the `saveToStorage` method and implement the way to retrieve the history from data storage in the `retrieveFromStorage` method. Developers may implement sql statements in `saveToStorage` method and `retrieveFromStorage` method to access the data storages immediately. But according to our experience, to use DAO can promote the reusability of the implementations of `Persistence` (i.e. the `DataBasePersistence`, `TextFilePersistence` and `XMLPersistence` classes).

5.8.2 Discussion of Persistence

In gerenal, the relational database systems are the most popular for database applications. However, in an object system, using relational database to store history may encounter some problems or difficulties. First, the mapping between database columns and the objects in WfMS is complicated. It is not easy to maintain the database tables. Second, it's also difficult to maintain the relationship between Objects in database record. The third problem is the identification of data. Data stored in database should have a unique indentification (primary key). How to generate the indentification and how to get the indentification when we want to retrieve or modifty the data, these two problems should be considered before we use database to store information.

Object-Oriented DataBase Management System (ODBMS) stores objects and it is much easier to maintain the relationships between objects. Therefore, some workflow systems use ODBMS to ease those problems caused by using relational database. For example, TriGSflow [20] and micro-workflow [8] use GemStone/S to store history as objects. GemStone/S adopts client-server architecture and provides some connectors to establish links between Smalltalk and GemStone class objects. For this issue, [42] also proposed a coexistence approach - using the RDBMS for legacy corporate data and the ODBMS for the process description and workflow status data.

Currently, we are trying to use the serialization for persistence. Java serialization is available in JDK1.1 and later version. It provides functions to serialize classes to serialized files and deserialize the serialized files to classes during the runtime of program. Java serialization preserves the variables, instances of internal objects and references of external objects. In JOO-WfMS, the related class instances will be

serialized when some events occur, e.g. the completion of activity or the occurrence of failure. Figure 65 illustrates the operations of seralization of Process and Activity instances in JOO-WfMS. When the execution state of process instance changes, the process instance fires the StateChangeEvent to its listener - the workflow engine, after receiving event from process, workflow engine will invoke the *Serialization Operator* to serialize the process instance. The serialization of activity instance also follows these steps.

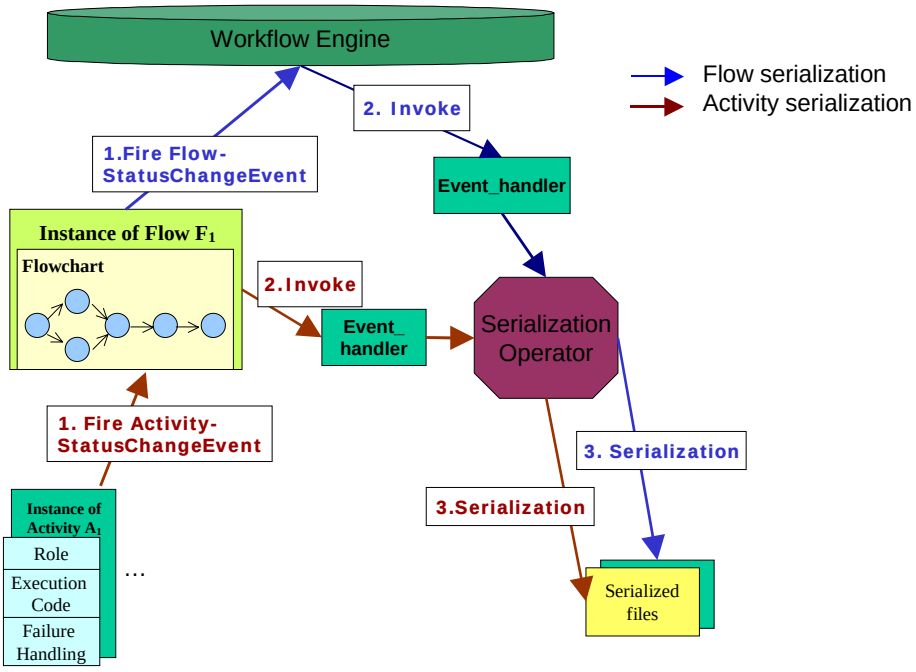


Figure 65: the operation of serialization in JOO-WfMS

Figure 66 shows the steps of deserialization of process instance. The workflow engine invokes the Serialization Operator the desialize the class instance from serialization files.

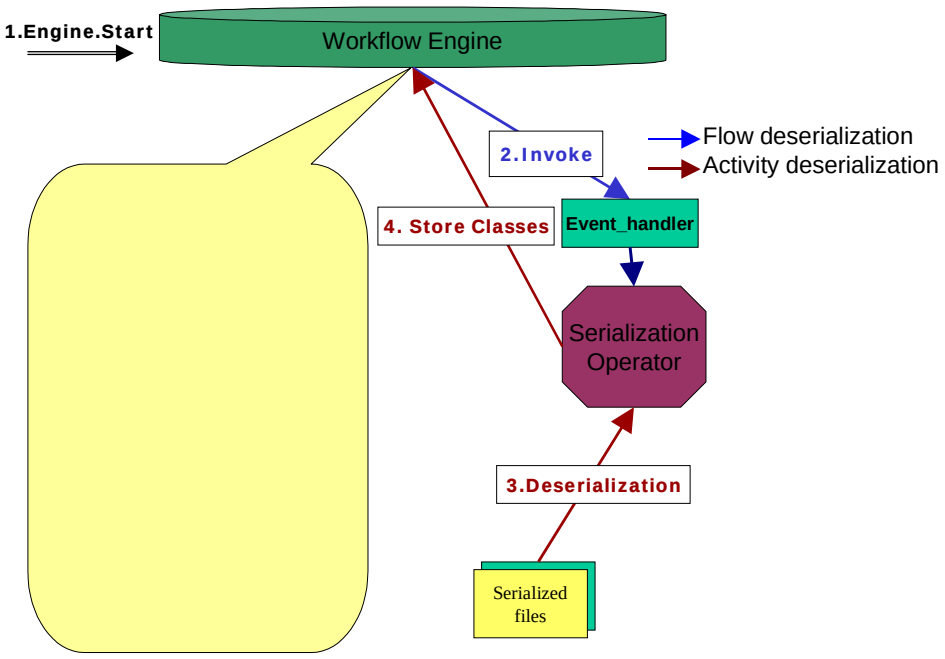


Figure 66: the operation of process deserialization in JOO-WfMS

Currently, we have implemented the serialization and deserialization of process and acitivity instances. Using serialization lets us need not to care about the problems caused by using RDBMS, but we think the operation of serialization and deserialization may have higher cost and may lower the system performance. Therefore, we will conduct a series of experiments to evalute the performance of serialization and deserialization. And how to query information from serialized files is another problem. We can use SQL statements to query data from database, but the way to retrieve informantion from serialized data is unknown now. In the future, we will try to find the way to query data from serialized files as easy as that in database. Moreover, we will try to integrate the ECA model introduced in [20] with persistence components. By using ECA model, the programmers can specify the “timing” to store history (e.g. store the information of an activity when its status is “Wait for user response”). We will introduce the details of ECA model in Section 6.3.