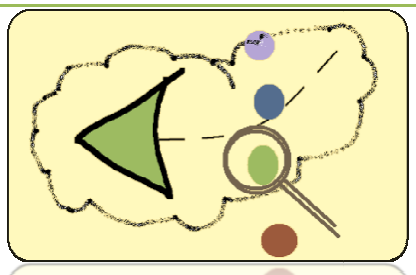


# NOVA WorkFlow

# User Manual

(version 2.0)

NOVA WorkFlow User Documentation
Version 2.0 / 2012 November
Author: Fazle Rabbi
Reviewer: Janet Norgrove



Centre for Logic and Information
St. Francis Xavier University
St Mary's Street
Antigonish, NS  Canada
B2G 2A5

Home page: www.logic.stfx.ca

## ACKNOWLEDGMENT

This work is sponsored by Natural Sciences and Engineering Research Council of Canada (NSERC), by an Atlantic Computational Excellence Network (ACEnet) Post Doctoral Research Fellowship and by the Atlantic Canada Opportunities Agency (ACOA) through the Atlantic Innovation Fund.

# TABLE OF CONTENTS

# Chapter 1

# INTRODUCTION

## Product Overview

Developed with understanding of compensable transaction and formal verification, NOVA WorkFlow is an innovative workflow modeling framework based on the Compensable Workflow Modeling Language (CWML)[1], a formal graphical language proposed by CLI. The framework consists of a graphical editor, a task specification editor, a code generator and a workflow engine.

The graphical editor provides visual modeling of workflow which ensures correctness by construction. The editor is developed as an Eclipse RCP plug-in[2], so one can make use of many UI features provided by Eclipse and install the editor in different OS platform.

A task specification editor has been incorporated in NOVA WorkFlow version 2.0 which allows writing T□ (T-Square[3]) code in eclipse editor. T□ is a domain specific language for workflow development. It incorporates following features: a) a simple syntax for i) writing procedural statements, ii) querying and manipulating ontologies, iii) designing a rich user interface (UI), iv) specifying access control policy, v) dynamically scheduling tasks; b) abstraction of communication details.

---

[1] For details on CWML, please refer to Fazle Rabbi, Hao Wang and Wendy MacCaull. "Compensable WorkFlow Net". The 12th International Conference on Formal Engineering Methods (ICFEM 2010).

[2] Eclipse, a popular and powerful Java IDE, is architected so that its components could be used to build just about any client application. The minimal set of plug-ins needed to build a rich client application is collectively known as the Rich Client Platform (RCP).

[3] For details on T□, please refer to Fazle Rabbi and Wendy MacCaull. “T-Square: A Domain Specific Language for Rapid Workflow Development”. ACM/IEEE 15th International Conference on Model Driven Engineering Languages & Systems (MODELS 2012), Innsbruck, Austria (September, 2012). Proceedings, Lecture Notes in Computer Science, Volume 7590. pp. 36-52.

We apply transformation methods, based on Xtend (http://www.eclipse.org/xtend ), to generate executable software from the abstract task specifications written in T□. Ensuring the correct transformation of T□ code to a software system gives us the confidence that the generated software components will produce correct workflows, a very essential feature for safety critical systems such as health care. This approach will help the customization of workflows by dramatically reducing the number of lines of code.

The workflow engine let you execute the workflow model built using the editor. The engine is developed using popular Spring (http://www.springsource.org) and Hibernate (http://hibernate.org) framework. The workflow engine can run in different platform with various database and web application servers. Fig 1.1 shows our approach of developing a workflow system.

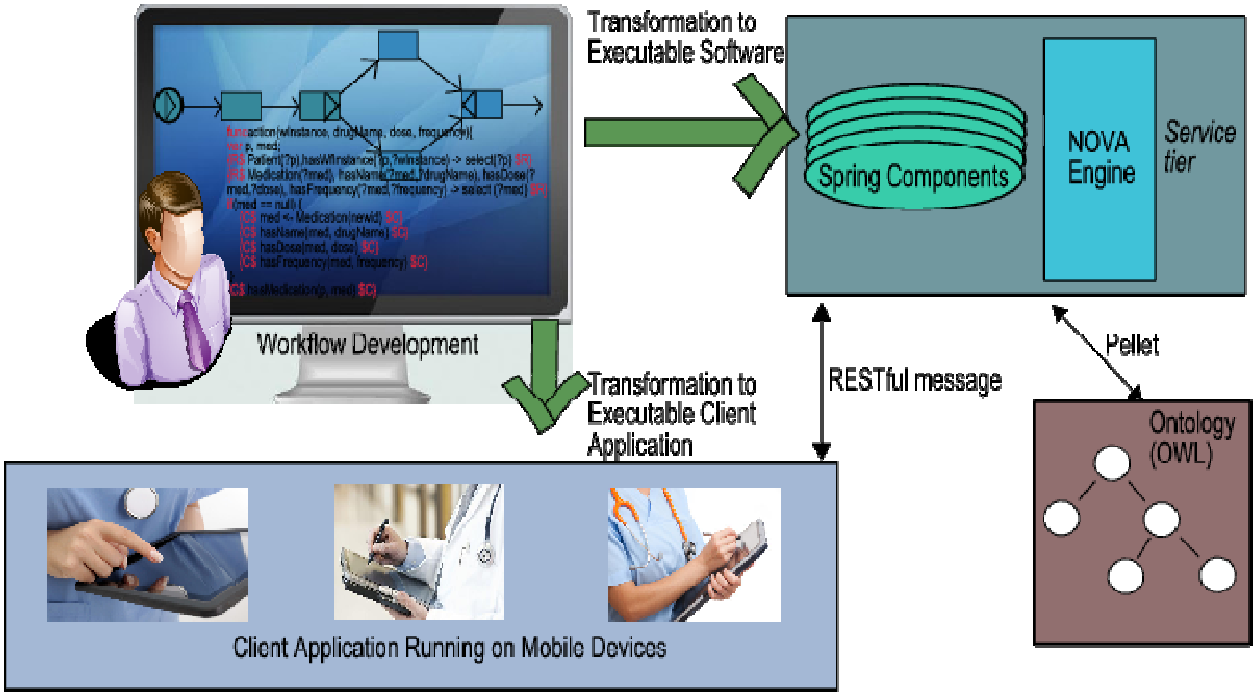



Fig 1.1: Workflow system development approach (NOVA WorkFlow 2.0)

# How to install

## Product Requirements

### Operating system (any one)

- ✓ SUN Solaris 2.6, 7, 8, 9 or 10[sparc]
- ✓ Linux- Red Hat Enterprise Linux/Fedora, Debian etc
- ✓ Windows 2000/2003 Server, Advanced Server
- ✓ Windows 2000/XP/Vista/2007

### Application Server (any one)

- ✓ BEA Weblogic Server 8.1/9
- ✓ Resin 3.0.x
- ✓ Apache Tomcat 5.0.x

### Database Server (any one)

- ✓ Oracle 9i Release 9.2
- ✓ MySQL 5
- ✓ Sybase 12.5 or higher
- ✓ PostgreSQL 8

### Java Devleopment Kit

- ✓ SUN JDK 1.6

### Open source software's

- ✓ Spring Framework 1.2
- ✓ Hibernate 3.5.4
- ✓ Eclipse Indigo 3.7
- ✓ Android SDK

## Starting the installation

Download and install Sun JDK 1.6 from http://java.sun.com and Android SDK from http://developer.android.com/sdk/index.html. Download eclipse with NOVA WorkFlow (Version 2.0) from http://logic.stfx.ca/software/nova-workflow/download/. Run eclipse and configure Android in Eclipse from Window-> Preferences.

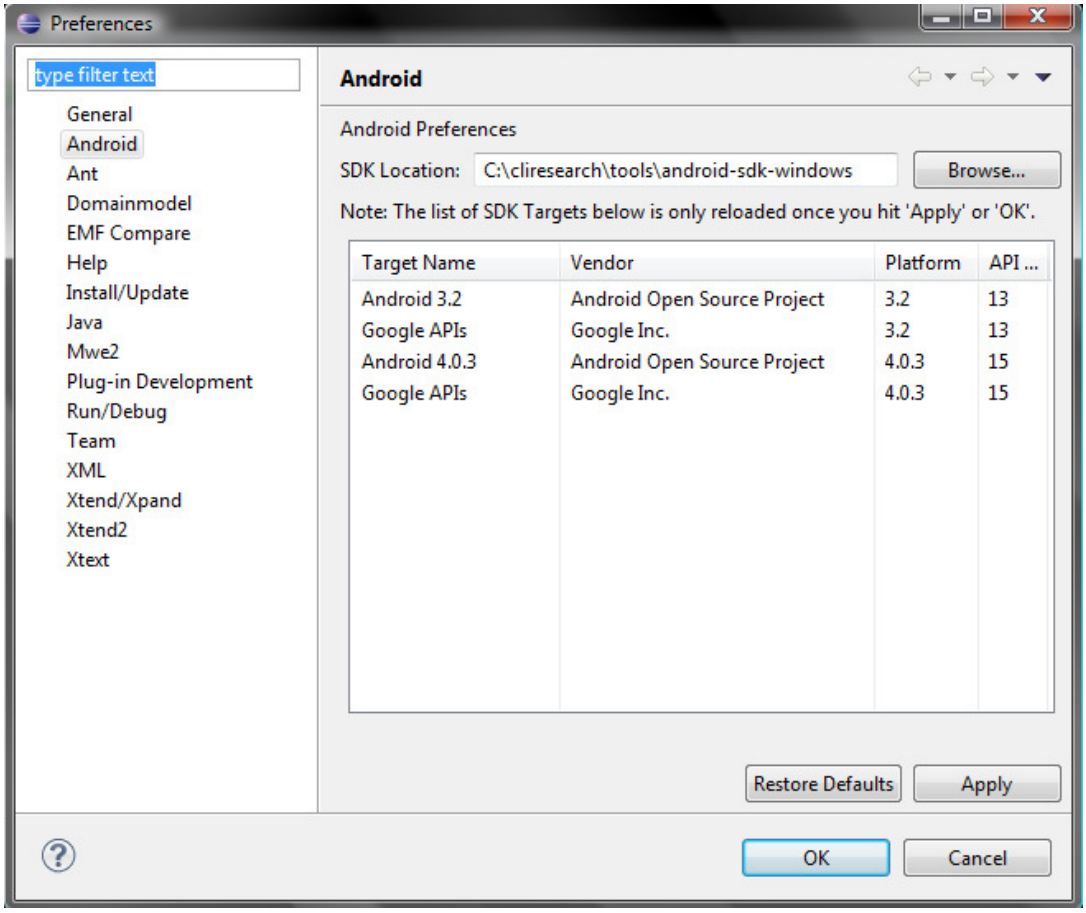



Fig 1.2: Configure Android in Eclipse

**Workflow components view**

To edit the workflow, open Workflow Components View. Workflow Components view can be found from Window->Show View-> Other -> CWML. You can also use the Outline view to get an outline of your workflow components.

**Where you will find**

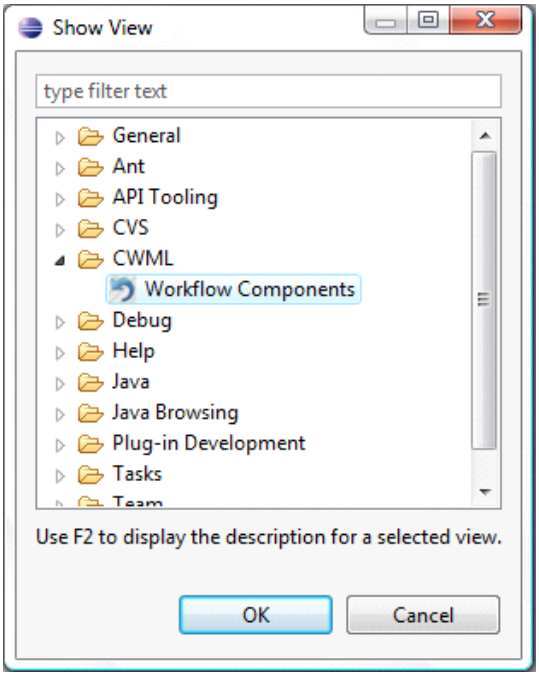



Fig1.3: Open Workflow Components View

Fig 1.4 shows the Workflow Components View. Using the tools you can easily edit your workflow.

### What you will see

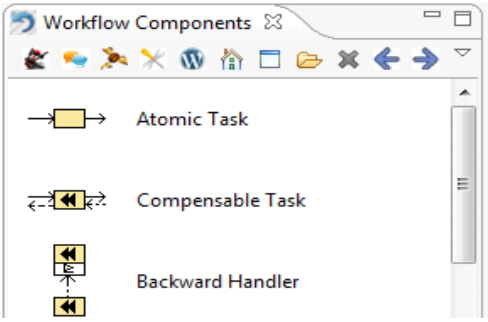



Fig 1.4: Workflow Components View

### Import a sample project

Open Eclipse and switch to the workspace directory inside the eclipse installation (/eclipse/workspace). Import the sample project named ***server*** which may be found inside the workspace directory.

### What you will see

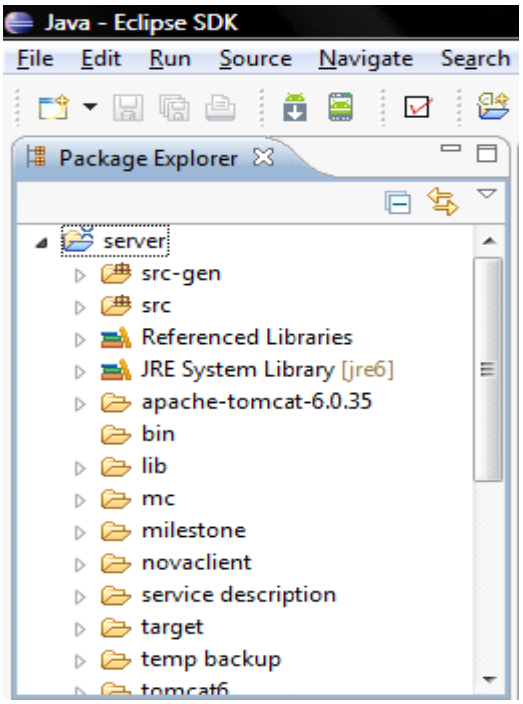



Fig 1.5: Directory Structure of the sample Project.

In this sample project ***server*** you will find a workflow named ***SCWorkflow*** (see Fig 1.6) under /src/seniorcare directory.

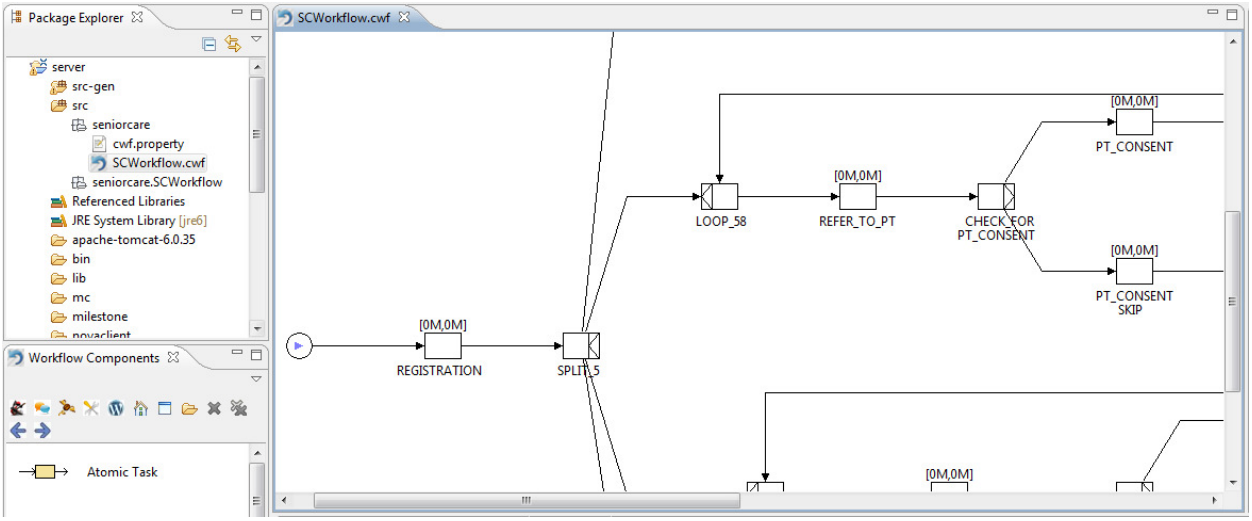



Fig 1.6: SCWorkflow.cwf

**Task specification file**

To write details of a task (i.e., task specification), select the task by left clicking on it and then click on the 'Task Property Settings' from the 'Workflow Components' View (see Fig 1.7). You will see a task property window. In the task property window you will find a check box to create a task specification file (see Fig 1.8).

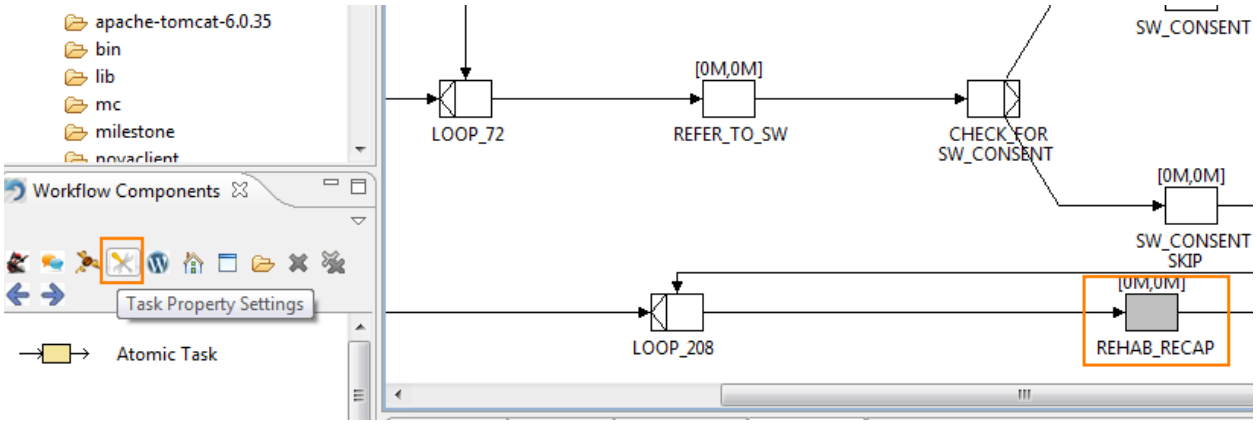



Fig 1.7: How to open a task property window

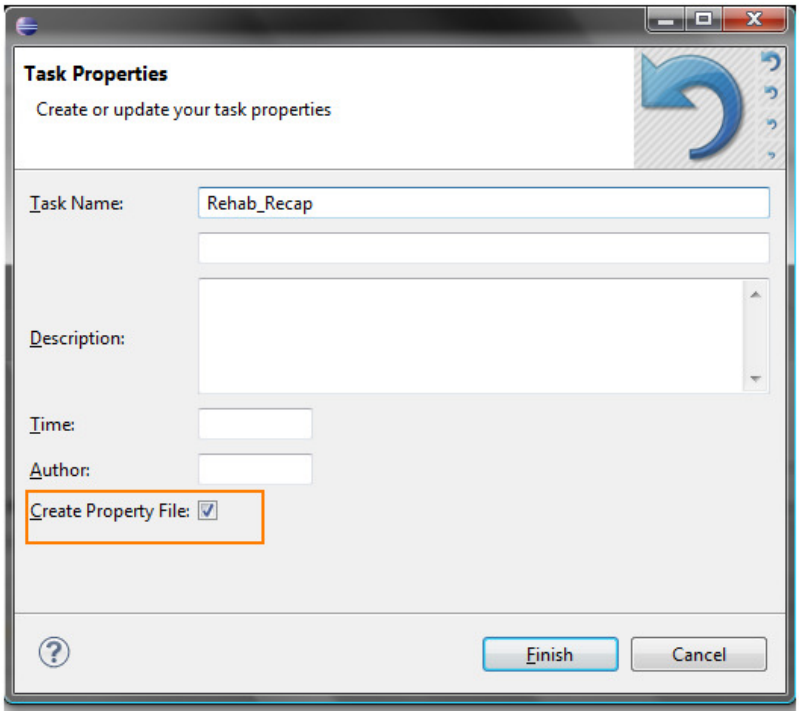



Fig 1.8: Task property window

Selecting the check box will create a task specification file if not already exists. To open the task specification file, click on the 'Open Property File' in the 'Workflow Component' view (see Fig 1.9).

**Where you will find**

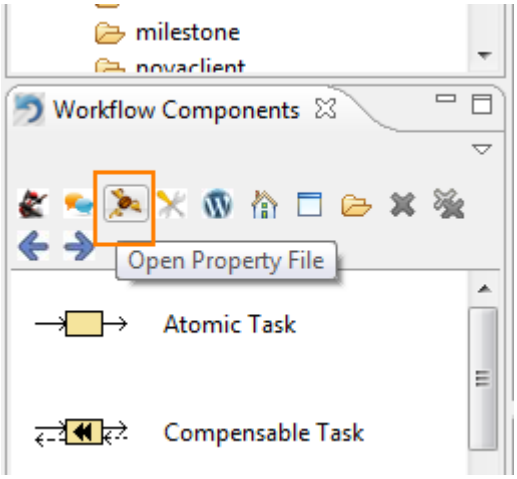



Fig 1.9: Open an existing task specification file

**Import the sample client application**

A sample android client application for the ***SCWorkflow*** has already been included under /workspace/server/novaclient directory. Import the ***novaclient*** application in your project by using eclipse import wizard.

**What you will see**

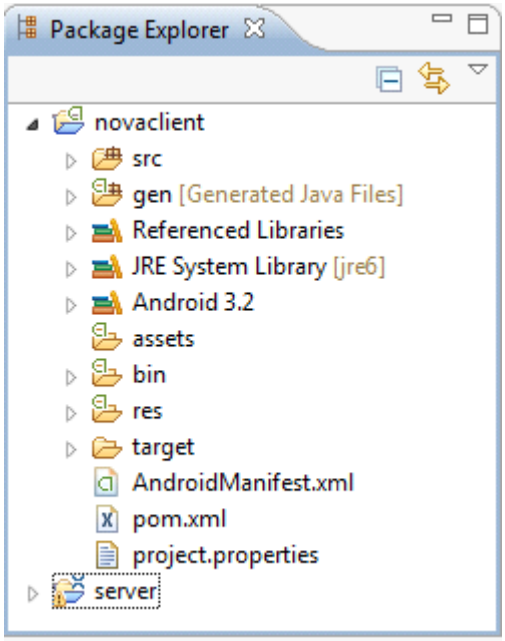
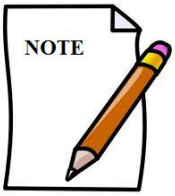



Fig 1.10 Sample android client application




A NOVA WorkFlow project consists of 2 separate projects:

1. Server
2. Client

The client application should physically locate under the server project directory to automatically generate the files required for the client application.

| | |
|---|---|
|   | To create a new workflow project, the best idea will be to download an empty workflow project from http://logic.stfx.ca/software/nova-workflow/download/, and then create a new package and workflows under the /server/src directory. |

## Create a new Workflow model

Create a ***package*** in your source (***src***) directory (or use an existing java package) where you want to store your workflow models. Right click on your ***package*** and select ***New -> Example***. Select '***New Workflow***' wizard and click next. You will see '***Create New Workflow***' Wizard. Enter the name of the workflow (file extension .cwf), author name. Select additional attributes for the workflow from drop down list. Attributes are described in the following table:

**Table1.1 Workflow attributes**

| Attribute | Value | Description |
|---|---|---|
| Root Net | True | During execution, a workflow with Root Net = True will start first. There can be only one workflow with Root Net = True in your workflow package. |
| | False | A workflow with Root Net = False is a subnet. A subnet can be decomposed by a composite task. During execution of a composite task, it is unfolded to a subnet |
| True Compensable | True | A True compensable workflow can hold only compensable tasks. A compensable task can only be decomposed to a True compensable workflow. |
| | False | A workflow with True Compensable = False can hold both compensable and uncompensable task. An uncompensable task can only be decomposed to this workflow. |

**What you will see**

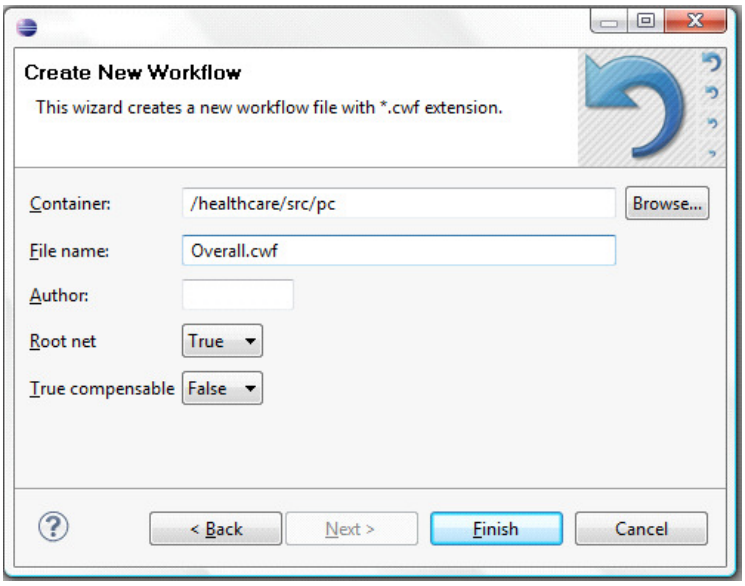



Fig1.11: Create New Workflow Wizard

Click ***Finish*** to create your first workflow model. An empty workflow model will open in the editor pane with an ***Input Condition*** and an ***Output Condition***.

# Chapter 2
# USING THE EDITOR

NOVA WorkFlow comes with a graphical editor for workflow modeling. The workflow you will make using this editor will be a structured workflow. The workflow model is stored in xml format.

## Insert an atomic task

To insert an atomic task in your workflow model use ***Pre-Selection*** and ***Post-Selection*** tools. The tools are available in ***Workflow Components*** View.

**Where you will find**

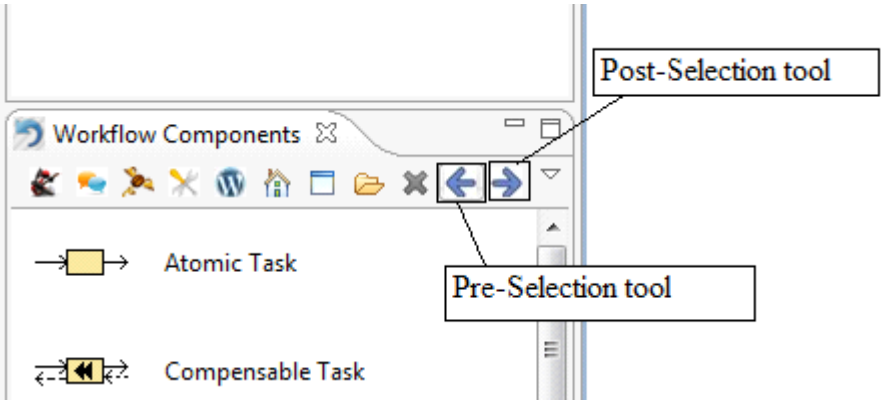



Fig2.1: Pre-Selection and Post-Selection tool

The ***Pre-Selection*** tool will change the color of a node to Green and ***Post-Selection*** tool will change the color of a node to Blue. After selecting two nodes by ***Pre-Selection*** and ***Post-Selection***, ***double click*** on ***Atomic Task*** from the Task list of ***Workflow Components***.

**What you will see**

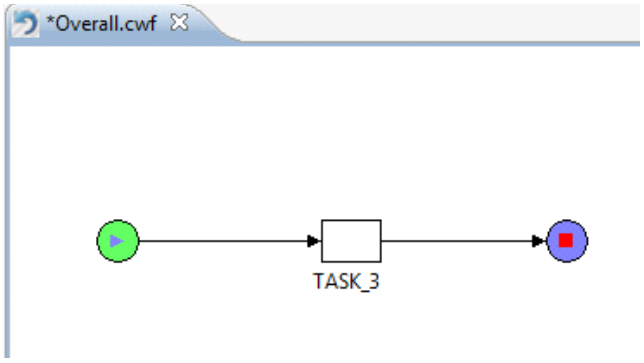



Fig2.2: Insert an Atomic Task

## Insert a split-join block

To insert a split-join block use the ***Pre-Selection*** and ***Post-Selection*** tool as before. Select two nodes where you want to insert your block, and then ***double click*** on the split-join block from ***Workflow Components***.

**What you will see**

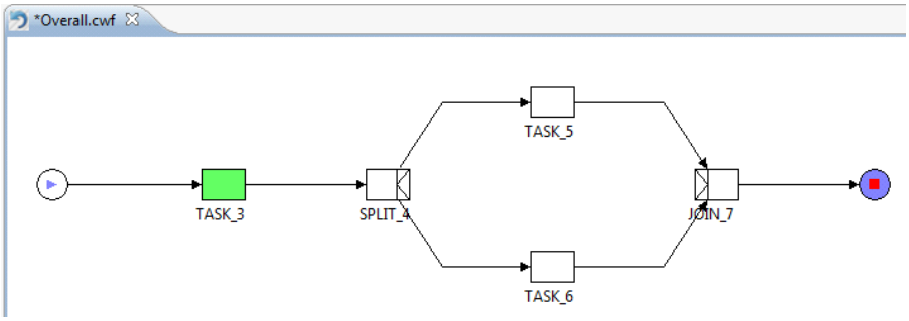



Fig2.3: Insert a Split-Join block

You can insert AND, XOR, OR, Parallel Composition, Internal Choice, Alternative Choice, Speculative Choice block in the same way.


**You cannot insert an uncompensable task or block inside a compensable block. If you try to insert, you will get an *Error message* "Invalid Selection: You cannot insert uncompensable task inside Compensable block".**

## Increase number of branches of Split-Join block

You can insert a new branch to a split-join block with an atomic task or another split-join block. To do this select the Split task of the block by ***Pre-Selection*** and the Join task by ***Post-Selection*** and ***double click*** on the task that you want to insert in a new branch from Workflow Components View.

### What you will see

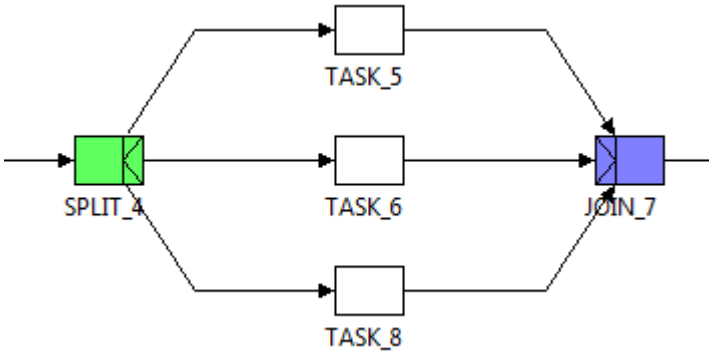



Fig2.4: Insert atomic task in a new branch

## Insert a Loop

To insert a loop around some tasks select two nodes using ***Pre-Selection*** and ***Post-Selection***. ***Double click*** on ***Loop*** from Workflow Components list.

### What you will see

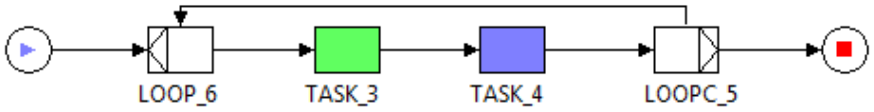
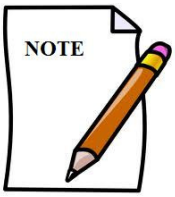



Fig2.5: Insert a Loop




**To insert a loop around a single atomic task, select the atomic task twice- first by *Pre-Selection* tool and then again by *Post-Selection* tool.**

## Edit a task

To edit a task, select it and then click on the ***Task Property Settings*** tool from Workflow Components View.

### Where you will find

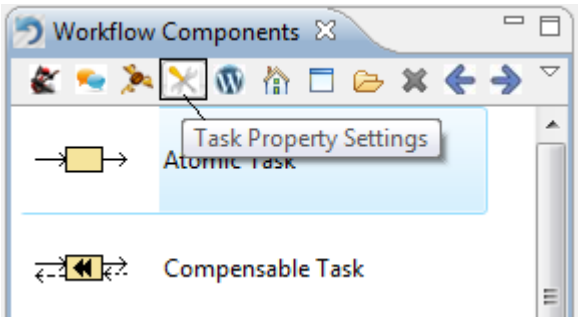



Fig2.6: Task Property Settings

This will open a Task Property dialog where you can edit task name, description, author name. There is a check box for creating Property file; this will be described in the next Chapter.

### What you will see

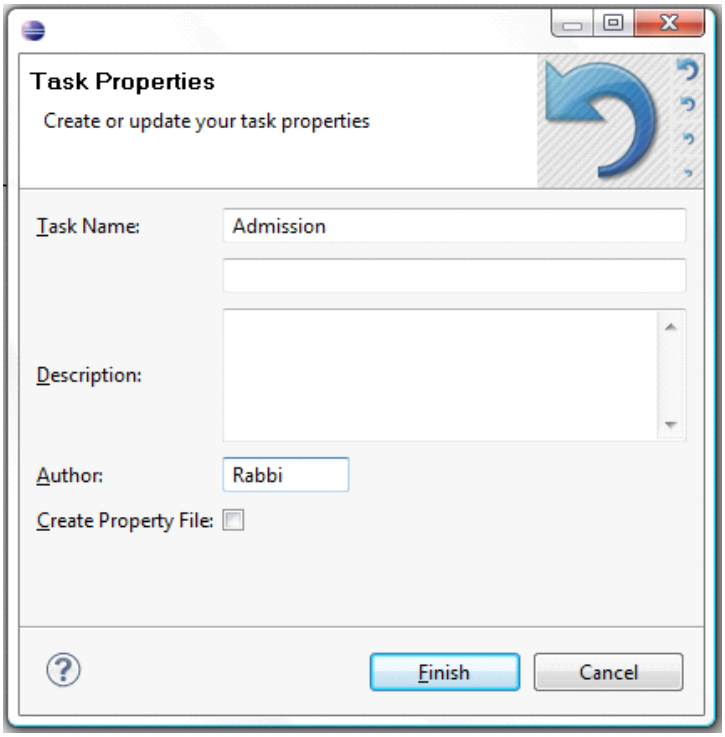



Fig2.7: Task Property Settings

## Delete a task

To delete a task or a split-join block, select the task by Pre-Selection tool, and then click on the delete tool from Workflow Components View. If you select a split task and click this tool, this will delete the whole block (split, join and all of its branches).

### Where you will find

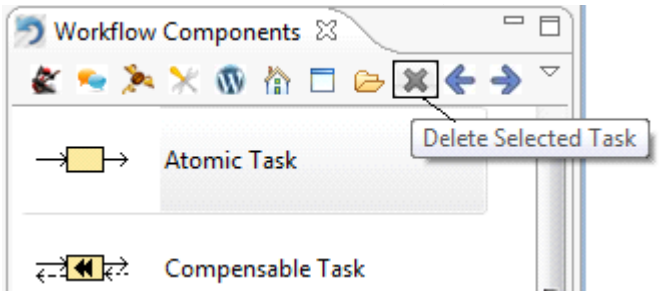



Fig2.8: Delete task tool

## Make Composite Task

Select the task (***Atomic*** or ***Compensable***) you want to make composite, and then click on the ***Make Composite*** tool from Workflow Components view. A ***Subnet*** workflow selection dialog will open from where you can assign the subnet.

### Where you will find

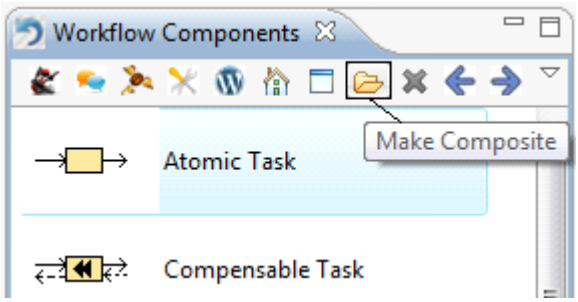



Fig2.9: Make Composite tool

### What you will see

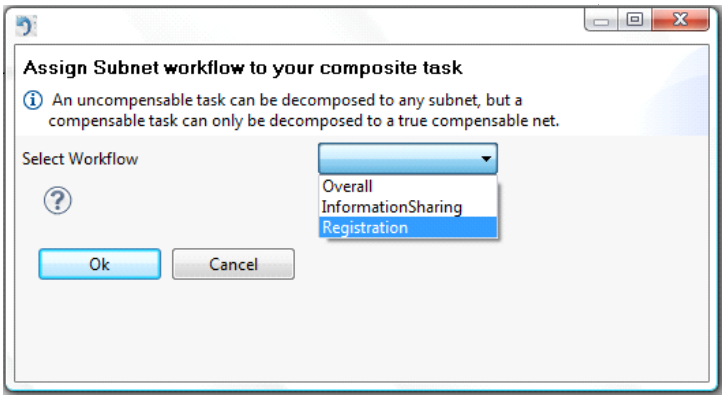
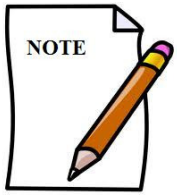



Fig2.10: Subnet Workflow Selection Dialog




**If you select a *Compensable* task to make it composite, you will only see Subnet workflows with *True Compensable = True*.**

# Chapter 3

# USING T□ TO WRITE TASK SPECIFICATION

T□ provides high level syntax for writing task specification. In addition T□ benefits from the use of ontologies which allows the user to infer knowledge. The idea of using an ontology instead of using a traditional relational database is the cardinal point of T□. Ontologies are knowledge representation technique which can represent many complex business rules using declarative specification. In contrast to traditional knowledge-based approaches, ontologies seem to be well suited for an evolutionary approach to the specification of requirements and domain knowledge. Software modelling languages and methodologies can benefit from the integration with ontology languages in various ways, e.g., by reducing language ambiguity, by enabling validation and automated consistency checking. Moreover intelligent applications may be built based on ontology reasoning.

## Overview of the features of T□

- ❖ Procedural language features
  - ✓ Inferred variables
  - ✓ If Else Statements
  - ✓ Arithmetic expressions
  - ✓ Loop
  - ✓ Function call (call by value / reference)
- ❖ Declarative language features
  - ✓ Supports SQWRL query format to query ontology
  - ✓ Simplified syntax to manipulate Abox (CRUD operations)
  - ✓ Integrated with an ontology reasoner
- ❖ User interface
  - ✓ Simplified syntax for UI creation
  - ✓ Simplified syntax for writing actions

## Procedural statements in T□

✓ Variables in T□ are inferred variables; variable types are determined from their use.

```
var a, b, c;
a = 10; b = 5;
a = b * c;
```

✓ Variables in T□ may be indexed as array indexes but a declaration of the size is not required. The size is adjusted dynamically at execution time.

```
var d;
d[5] = 100;
d[3] = 50;
```

- ✓ In T□, procedures may be invoked by `call by value' or `call by reference'.

```
func factorial(n){
        if(n <= 1 ){
                return n;
        }
        else{
                return factorial(n)* factorial(n-1);
        }
}
```

- ✓ In T□ syntax for the Assignment operation, If-Else statements, For-loops, etc., are similar to that for the C family of languages.

```
func sum(numbers){
        var sum = 0;
        foreach( n in numbers){
                sum = sum + n;
        }
        return sum;
}
```

- ✓ In T□ some utility procedures such as *size*, *today*, *currentTime*, *date*, *month*, *year*, *time*, and *tokenize* have been incorporated to deal with strings, arrays, dates and times.

### Query and manipulate ontologies

Ontologies allow data and rules to be organized efficiently so as to permit the calculation (i.e., inference) of implicit knowledge from explicit information. Using ontologies to drive workflows allows for a more compact representation of the workflow and changes made in an ontology (which are often simple to implement) can avoid the need to change the workflow (which can be more complicated). An important aspect in the design of T□ was the facility to query and

manipulate ontologies. It provides four different tags to perform Create (C), Read (R), Update (U), and Delete (D) operations (CRUD operations) in an ontology.

### Query an ontology

One can perform queries combining concepts and facts from the Tbox and/or Abox. The Tbox describes conceptualizations and contains assertions about concepts such as subsumption (Man is a subclass of Person). The Abox contains role assertions between individuals (hasChild(John, Mary)) and membership assertions (John : Man).

T□ allows us to write queries in the easy-to-use SQWRL format. Similar to the `select' operator of SQWRL, the `select' operator in T□ takes one or more arguments, which are typically variables used in the pattern specification of the query. A particular value may be passed as a query criterion; if a variable is used in an ontology query without a leading question mark (?) then the value is read by the query engine. The following query, written in the SQWRL format, retrieves all persons in an ontology with a pain intensity that is greater than 5, together with their pain intensities:

```
var  p, pain, v;
v = 5;
{R$  Patient (?p), hasPain(?p, ?pain), greaterThan(?pain, v) → select(?p, ?pain)  $R}
```

The query engine will populate the variables passed as arguments of the `select' operator. Selected results may be sorted in ascending (descending) order by the `orderBy' (`orderByDescending') operator.

### Create a new fact

To create a new instance/individual or relation in the ontology Abox, the OntAssertion statements may be used directly from T□. The following OntAssertion statements create a new `Patient' individual and inserts a data property for the relation `hasPain'.

```
var  p;
{C$  p := Patient(“Alex”) $C}
{C$ hasPain (p, 6) $C}
```

Note that a reference of the newly created Patient individual is assigned to the variable `p'. An individual may be created with an auto-incremented identity (id) if in the ontology there exists a data property named `hasId', where the domain of `hasId' is `Thing' and range is `Long' data type. The following code shows how to create a new Patient individual with an auto-incremented id.

### Delete a fact

OntDel statements may be used to delete an individual or a relation from an ontology ABox. The following code shows how to delete a Patient with id=1010.

```
var  p, pid;
pid = 1010;
{R$  Patient(?p), hasId(?p, ?pid) → select(?p) $R}
{D$ Patient(p) $D}
```

In this code fragment, a search operation is performed on an ontology for a Patient individual with id=1010 and a reference is retrieved; the Patient individual's reference is then passed as an argument to the delete operation.

### Update a fact

To update a data property or object property of an individual, OntUpdate statements may be used. Following code fragment shows the use of an update operation. This code fragment updates the ages of all patients whose birthday is today.

```
var  p, P, bDate, Age, age, newAge, cDate;
cDate = today();
{R$  Patient(?P), hasBirthDate(?P, ?bDate), isEqual(?bDate, cDate), hasAge(?P, ?Age) →
        select(?P, ?Age) $R}
foreach (p in P, age in Age){
        newAge = age + 1;
        {U$ hasAge(p , age => p, newAge) $U}
}
```

### Design User Interface

To print a text or a number in a UI, the *getLabel* procedure may be used. The *getLabel* procedure produces a `Label' view component in the UI. One can pass either a string literal or a variable as argument of the *getLabel* procedure. If a variable is passed, the variable is bound to a `Label' view component. Whenever this variable is updated, the change is reflected in the `Label'.

```
var    wid;
wid = 112;
getLabel(‘‘WorkflowInstance:’’);
getLabel(wid);
```

This code fragment produces two labels; during execution, the first label will display the text “Workflow Instance:” and the second label will display the number `112'.

The *getText* procedure produces a ‘Text Field’ view component. A ‘Text Field’ is a common UI component to take user input. The *getText* procedure can take one or two arguments: i) a string to produce a label, and ii) a variable (optional) to display the initial text in a ‘Text Field’. A destination variable name after the symbol ‘>>’ is required for a *getText*. The user input is captured by the destination variable. Optionally, some statements (also known as action statements) may be written inside curly braces after the destination variable name of a *getText* procedure. These action statements will be executed when a user finishes her entry into the ‘Text Field’. The following code fragment uses the *getLabel* and *getText* procedures:

```
var    hospitalName, displayText;
displayText = “No Input”;
getText(‘‘Enter Hospital Name:’’) >> hospitalName{
          displayText = “Hospital:  ’’+ hospitalName; };
getLabel(“Entered Text: ’’, displayText);
```

This produces a ‘Text Field’ where the user will enter text input; the entered text will be stored in a variable named ‘hospitalName’. As soon as the user finishes entering text into the `Text Field' the action statement (enclosed with a curly bracket) will execute and assigns a value entered by the user to the variable named ‘displayText’. Since the variable ‘displayText’ is bound to a ‘Label’, when its value changes, the ‘Label’ view component will be updated and will

display the hospital name entered in the 'Text Field'. The *getPassword* procedure produces a 'Password Field' view component where the input characters are displayed as dots on the screen. The other functionalities of a 'Password Field' are the same as those for a 'Text Field'.

The *getInteger* procedure is similar to the *getText* procedure; this also produces a 'Text Field' to take input from the user, but the difference is that only numbers are allowed here. The following code fragment gives an example:

```
var    basicPay, hourlyPay, totalHr, totalSalary;
basicPay = 14; totalHr = 0; totalSalary = 0;
getInteger(''Hourly Payment: $'', basicPay ) >> hourlyPay{
totalSalary = hourlyPay * totalHr; };
getInteger(''Total Hour Worked: '') >> totalHr{
totalSalary = hourlyPay * totalHr; };
getLabel("Total Salary: $", totalSalary );
```

The first 'Text Field' will display the value of the 'basicPay' variable which is '14'. The user may change it by inserting a different number; the entered number will be stored in the variable named 'hourlyPay'. The user can also enter the total hours worked in the second 'Text Field'. Whenever the user finishes entering numbers in either of the 'Text Fields' the total salary is calculated and displayed in the UI by a 'Label'.

The *getDouble* and *getDate* procedures are similar to the *getInteger*; here the user can enter a floating point number and date respectively. The *getTextMultiple* procedure is similar to the *getText* procedure but it produces a 'Text Area' (for multiline text input) instead of a 'Text Field'. The *getDate* procedure is similar to the *getInteger* procedure but here the user enters a date in a 'Text Field'. The *getBoolean* procedure takes one argument as input to display a title for a 'Check box' (a view component to select or de-select an item). The user may select or de-select the 'Check box' and a true or false value is assigned to the associated destination variable of a *getBoolean* procedure. If action statements are written for a *getBoolean* procedure, they will be executed  after the user selects or de-selects a check box item.

```
var    painCrisis;
getBoolean(''PainCrisis:'') >> painCrisis;
```

The above code fragment will display a 'Check box' in the UI. The destination variable of the 'Check box' is 'painCrisis' which will be assigned with a 'true' or 'false' value depending on the selection of the 'Check box'. Since the 'painCrisis' variable is bound to a 'Check box' view component, if the value is changed from another portion of the procedure, it will be reflected in the UI. This feature may be useful to display a form to update existing information. For example if we want to display a patient's existing Pain Crisis information and allow the user to modify it, we can use the following code:

```
var    painCrisis, id, p;
id  = 1010;
getBoolean(''PainCrisis:'') >> painCrisis;
{R$  Patient(?p), hasId(?p, id), hasPainCrisis(?p, ?painCrisis) → select(?painCrisis)  $R}
```

The *getOne* procedure is used to select one item from a list of items. This procedure will either display a 'Drop Down' view component (if one source variable is provided as argument) or a 'Table' with 'Radio buttons' (if more than one source variable is provided). The user cannot select more than one item from the displayed list. A destination variable name is required for a *getOne* procedure where the selected item (user input) will be stored. Optionally another destination variable name may be mentioned to store the position of the item selected from the source variable. If action statements are given for a *getOne* procedure, they will be executed as soon as the user selects an item. In the following example a list of countries is retrieved from an ontology by performing the read operation. A *getOne* procedure is used to display the list of country in a 'Drop Down'. Another *getOne* procedure is used to display a list of provinces in another 'Drop Down'. Since the provinces are different from one country to another, on the selection of a country, a further query is performed on the ontology to retrieve related province information; this is done in the action statements. The 'province' variable is bound as the source variable with the second *getOne* procedure; as a result, if provinces' information were updated they will be reflected in the 'Drop Down'.

```
var    c, country, province, selectedProvince;
{R$ Country(?c) → select(?c) $R}
getOne(''Country:'', c) >> country{
        {R$  Province(?province), hasCountry(?province, country) → select(?province)  $R}
};
getOne("Province:", province) >> selectedProvince;
```

Note that the 'source' variable fills a 'Drop Down' view component but if we want to display one item from the items available in the 'Drop Down' we may use the 'destination' variable. For instance, in a patient's admission record update form, we want to display the patient's existing province in the 'Drop Down'; this can be achieved by assigning the name of the province to the destination variable.

The *getMultiple* procedure is similar to the *getOne* procedure but here the user may select more than one item from the source variable(s). The values of the source variable(s) are either displayed in a list of Check Boxes or in a `Table' with `Check Boxes' (for more than one source variable).

```
var ref, refName, referral;
{R$ Referral(?ref), hasName(?ref,?refName) → select(?refName) $R}
getMultiple (''Referral:'',refName) >> referral;
```

This code shows a use of a *getMultiple* procedure. A list of referrals is retrieved from an ontology which is used as the source which the *getMultiple* procedure uses and displays in the UI. From the displayed referral information, the user might select more than one referral. The selected referrals will be stored inside the destination variable named 'referral'.

The *getButton* procedure produces a button in the UI. When a button is pressed, the statements associated with it are executed. For example, if we want to calculate the strength of a given password then a button may be used to do the activity.

```
var password, result;
getPassword( ''Enter Password'' ) >> password ;
getButton( ''Check Password Strength'' ) {
        result = checkPwStrength( password );
        print(result);
}
```

When the button "Check Password Strength" is pressed it invokes a procedure named *checkPwStrenth* at the server with 'password' as argument and prints the result in the UI. T□ provides two procedures to arrange the view components in the UI; namely *openLayout* and *closeLayout*. The *openLayout* procedure takes an integer parameter which indicates the number of columns. All view components mentioned after a *openLayout* procedure will follow

this arrangement. A *closeLayout* procedure stops putting view components in the order started by an *openLayout* procedure. An *openLayout* procedure can be nested with another *openLayout* procedure; in this way a complex table layout structure may be achieved.

## Server vs. Client side code

Workflow developers are allowed to write specifications for tasks using T□ syntax. A developer may write procedures inside a task specification file. Since the specifications are translated into executable java code, the compiler scans the code for procedures named ***view()*** to generate the UI interfaces (i.e., Forms); other procedures are used to generate server side components. Below is a description of how different procedure names are interpreted in NOVA WorkFlow.

**Table 3.1: List of procedure names and how they are interpreted in NOVA WorkFlow**

| Procedure name | Parameters | Description |
|---|---|---|
| view | 0 | All the statements or code written in a procedure named view() will be transformed to a UI-Form. |
| action | User defined | If a view() procedure is provided for a task, the action() procedure is invoked from the vew() procedure by a submit() function call. Workflow developers are allowed to specify the number of parameters for such an action() procedure. |
| | 0 | If a view() procedure is not provided for a task then the action() procedure is invoked by the workflow engine as soon as the task becomes enabled. The workflow engine will invoke the action procedure with 0 parameters. |
| getDelay | 0 | Workflow developer may specify the delay time constraints for a task in this procedure. This method should return number values. |
| getDuration | 0 | Workflow developer may specify the duration time constraints for a task in this procedure. This method should return number values. |
| getUserRoles | 0 | Workflow developers may specify the name of the roles who are allowed to access the task. This method should return array of strings. |

| | | |
|---|---|---|
| accessPolicy | 0 | The access policy for a task may be specified in this procedure. Before invoking the action() procedure the workflow engine will consult this procedure. If this procedure returns false, the access will be rejected. |
| getBranchCondition | 2<br><br>( wInstanceId, brNo ) | The branching conditions for XOR, OR, Loop Split tasks are written in this procedure. The workflow engine will consult this procedure to determine which branches are active/inactive. |
| preCondition | 0 | If a precondition() procedure is provided it will be consulted to determine if a task is enabled or not. If this procedure returns false, the task will not be displayed in the worklist window. |

# Chapter 4

# USING THE WORKFLOW ENGINE

NOVA WorkFlow engine is a flexible workflow engine developed in the Spring & Hibernate framework. You may deploy the engine into different application servers. However the sample application shipped with NOVA WorkFlow is using Tomcat servlet container.

## Configure your project for deployment

A database script (dbcwf.sql) has been provided in the sample application which includes the sql statements to create few tables in a database. The script may be found in /eclipse/workspace/server/ directory. You have to import the script in a relational database: the script will create a database named 'dbcwf' and will create the required tables for the workflow engine.

Click on the 'Workflow settings' button in the 'Workflow Components' view (ee Fig 4.1). This will open a 'Workflow Settings' window (see Fig 4.2).

**Where you will find**

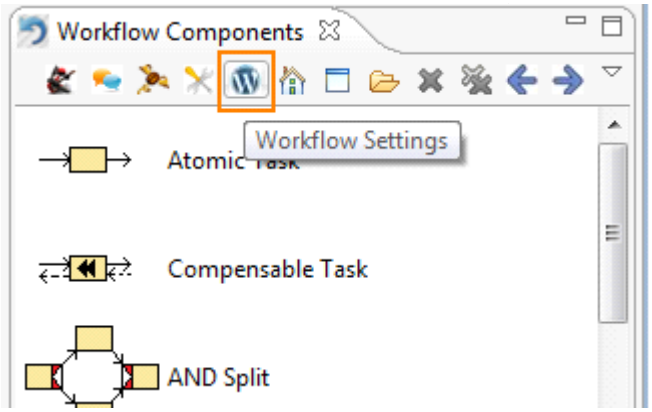



Fig 4.1: Workflow Settings button

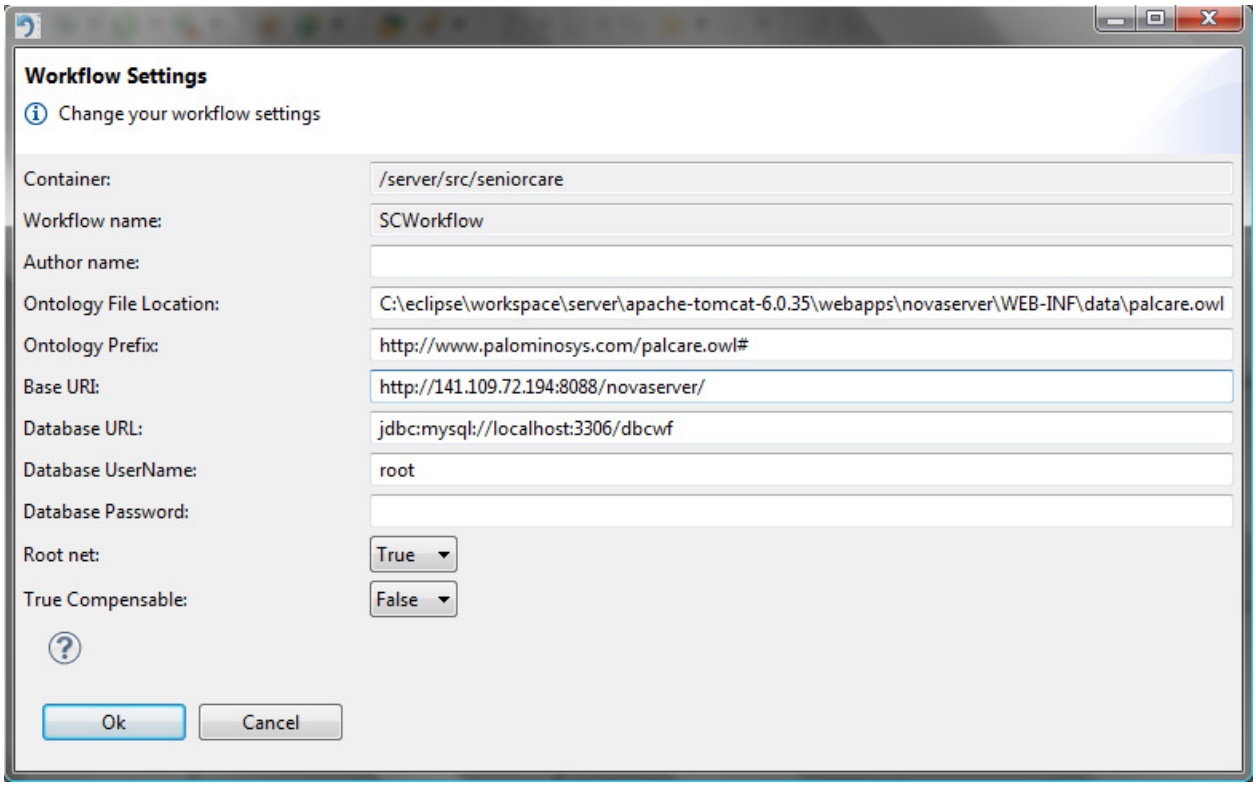



Fig 4.2: Workflow settings (Deployment configuration)

You have to provide the following details to configure the workflow engine and the client application:

| Parameter | Description |
|---|---|
| Ontology File Location | The workflow engine will try to access (read and write) an ontology file. As you already know that NOVA WorkFlow uses an ontology to persist information, the engine will require an owl ontology file. For the sample application, an ontology file has been provided in the following location:<br><br>/eclipse/workspace/server/apache-tomcat-6.0.35/webapps/novaserver/WEB-INF/data/palcare.owl |

| | |
|---|---|
| Ontology Prefix | The prefix that should be used while you are querying and manipulating your ontology using T□. In Chapter 3 you have seen that we have talked about a concept 'Patient' but typically in an ontology, it has a prefix, such as, http://www.palominosys.com/palcare.owl# |
| Base URI | The client application will use RestFul webservice to communicate with the workflow engine. The web address of the server has to be specified in this field. If you are using the sample application, you may write down the following URI in this field: http://<yourip>:8080/novaserver/ |
| Database URL | The states of the workflow will be stored in a database. And the workflow engine will use hibernate to access the database. |
| Database UserName | The username of your relational database |
| Database Password | The password of your relational database |

## How to deploy

Click on ***Create Service Class*** tool from your Workflow Components view.

## Where you will find

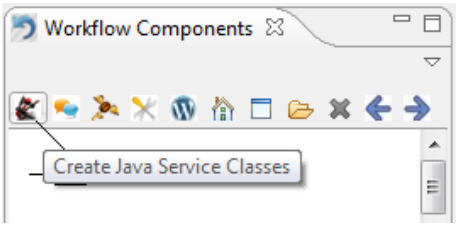



Fig4.3: Tool for generating Service Class

For each of the task of your workflow a service class will be generated. Some hibernate mapping files; an applicationContext.xml file will be generated that is required for the execution of workflow engine.

By clicking the tool will also configure the client application as well and everything should be ready to start. Start the tomcat server first by executing startup.bat / startup.sh which may be found from eclipse/workspace/server/apache-tomcat-6.0.35/bin directory.

You can either start the client application (***novaclient***) in an android device or an android emulator.

## Play with the application

The client application has a default user management screen. You may create new users, roles and assign roles to users. The default administrator's userid and password is superadmin/123456. You can change it from the user management control panel if you wish. To start working with your workflow, you will have to create a workflow instance. Once you have created a new workflow instance, you will see the list of available tasks in the ***worklist*** window. From the ***worklist*** window you can access the tasks. If you submit any information from a Form, that information will go to the server and the action() procedure will be invoked. The workflow engine will update the task status and will update the worklist window as well.