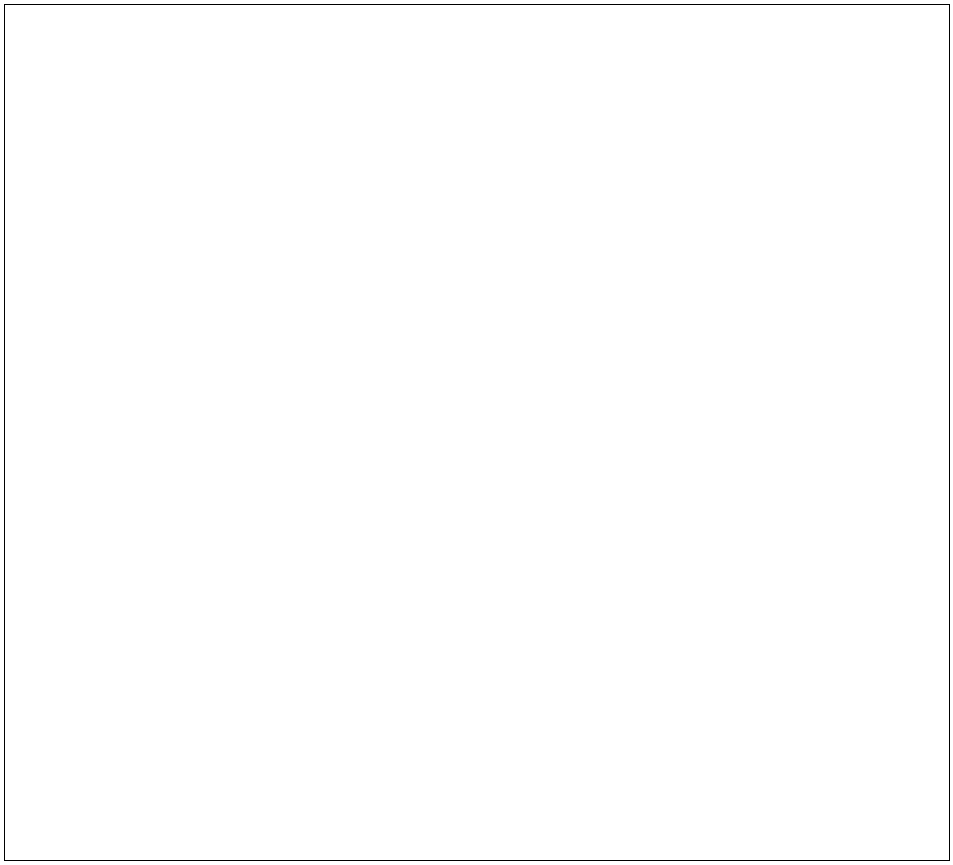

zen Platform 2.0 Tutorial

Table of Contents

1 Overview

This tutorial explains step by step how to build a web application with the *zen Platform,* on the basis of an online banking example. The example application will provide usual functions like checking the balance, filling in and inspecting transfers, just like a real world banking application would do. The needed functionality will be added and explained step by step.

2 Installation

The following chapter describes how to install the Eclipse development platform and the *zen Developer* plugin for developing applications on the *zen Platform.*

2.1 System requirements

The minimum requirement in order to install Eclipse and the *zen Developer* is a *Java 2 SDK, Standard Edition* (J2SE SDK) version 1.4.x or higher.

If you haven't installed this JDK yet, you can download it from

http://java.sun.com/j2se/1.4.2/download.html

for free. Please follow the given instructions.

2.2 Installation of Eclipse

The *zen Developer* is a plugin for the freely available development environment Eclipse. You need an Eclipse-SDK version 2.1.2 or higher. Eclipse 3.x is supported as well.

Installation

If have not installed Eclipse yet, please download the appropriate version from:

http://www.eclipse.org/downloads/

We recommend the so called “Latest Release”. You can find them directly following these links:

2.1.3: http://download.eclipse.org/downloads/drops/R-2.1.3-200403101828/index.php
3.0.1: http://download.eclipse.org/downloads/drops/R-3.0.1-200409161125/index.php

Eclipse is distributed as ZIP-File without an installer. Simply unpack the contents to your favored directory.

Starting Eclipse

Eclipse is started using the executable “eclipse”, located in the eclipse directory where you have unpacked the contents of the ZIP-File.

> e3 The Eclipse 3 GUI is somewhat different from Eclipse 2. This tutorial is based on Eclipse 2, differences are shown in separate chapters or boxes. If you like to use Eclipse 3 but want to use the appearance of Eclipse 2, you can do so by choosing *Window->Preferences* and selecting *Current Presentation* : *R21Presentation* from the *Workbench->Appearance* section.

2.3 Installation of zen Developer for Eclipse 2

Please launch Eclipse now. If this is your first Eclipse launch, you will see a welcome page with some information about Eclipse.

To install the *zen Developer* plugin, activate the *Install/Update-Perspective* by choosing *Help->Software Updates->Update Manager* from the menu.

The easiest way to install the *zen Developer* plugin is by using the zeos-update-site. If you can't establish a direct internet connection (port 80) from your computer, you can download the installation package separately. In this case follow the instructions layed out in section *Installation within Eclipse 2 as separate download* further below.

Installation within Eclipse 2.x using the update site (recommended)

On the lower left hand side of the *Feature Updates View* (see Figure 1) click with the right mouse button and select *New->Site Bookmark...*. Now you can enter the update site details for the *zen Developer* plugin, needed for automatic download and installation.

- Enter `zeos` as name for this Update-Site.
- Enter the following address as URL `http://www.zeos.de/update/zen/` .
- Close the dialog by selecting *Finish*.

The zeos update site is now visible in the *Feature Updates View* . If you expand the zeos branch, the installation tree will look like this:

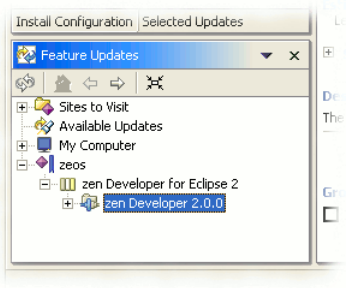

Figure 1
Bookmark for the zeos update site

Please continue now with section *Starting the installation within Eclipse 2.x*.

Installation within Eclipse 2 as separate download (alternative)

If you can't install *zen Developer* using the update site, please download the complete installation package (ZIP) from

http://www.zeos.de/support/download/

and unzip the files in any directory. Within the *Feature Updates View* of Eclipse you can now browse your hard disk using the *My Computer* tree. Please locate the directory containing the unzipped files. This folder should now show up with *zen Developer* for Eclipse 2 similar to the figure above. After finishing the installation, the directory containing the installation files may be removed.

Starting the installation within Eclipse 2.x

Select the node below *zen Developer for Eclipse 2* labeled *zen Developer 2.x.y* (where x and y represent the current version)

To actually start the installation, please click on *Install Now* in the right hand side of the window. The installation wizard will guide through the remaining steps. Please don't alter the *Install Location* during this procedure. By clicking *Finish/Install*, the *zen Developer* plugin is loaded from the zeos website (or your local disk) and installed into the following folders below your Eclipse installation:

- eclipse/features
- eclipse/plugins

Please restart Eclipse after the installation.

License key

If this is your first installation, Eclipse reminds you that the *zen Platform license key* has not been installed so far. After this dialog box you will see the Eclipse welcome page of your *zen Developer*.

Please copy the license file you received from us into the following directory below your Eclipse installation:

plugins/de.zeos.zenit.lib.runtime_2.x.y

just like mentioned on the Welcome-Page. x and y stand for the highest current installed version of *zen Developer.*

If you haven't received a license file so far, you can obtain one from:

http://www.zeos.de/support/download/license.php

After these steps, Eclipse and the *zen Developer* plugin are installed and operational. You can find the release notes in the directory *plugins/de.zeos.zenit_2.x.y* of your Eclipse installation.

2.4 Installation of zen Developer for Eclipse 3

Please launch Eclipse now. If this is your first Eclipse launch, it will start with the *Workspace Launcher.* You have to select, where Eclipse will store your working data and configuration. By selecting *Use this as the default and do not ask again* you may set the selected directory as default directory. After that, you'll see the a welcome page within the Eclipse workbench.

To install the *zen Developer* plugin, please start the installation process by selecting *Help->Software Updates->Find and Install* from the menu. The *Install/Update* wizard is shown. Please select *Search for new features to install.*

Installation within Eclipse 3.x using the update site (recommended)

On the subsequent page please select *New Remote Site...* and enter the following details for the update site used to download and install the *zen Developer* plugin automatically:

- Enter `zeos` as name for the update site.
- Enter `http://www.zeos.de/update/zen/` as URL
- Close this dialog by clicking *OK*.

The zeos update site is now displayed within the wizard. Expand the zeos node to see the node *zen Developer* for Eclipse 3. Please proceed with the section *Starting the Installation within Eclipse 3.x*

Installation within Eclipse 3 as separate download (alternative)

If you can't install the *zen Developer* plugin using the zeos update site, please download the complete installation package (ZIP) from

http://www.zeos.de/support/download/

and save it to your local hard disk. Use the button *New Archived Site* to select this ZIP file. After the installation has finished, you may remove this file.

Starting the installation within Eclipse 3.x

Now select the node labeled *zen Developer for Eclipse 3* below the *zeos* node within the installation wizard. By clicking on the button *Next* you get to the next page where you can select the *zen Developer* feature. Follow the remaining steps of the wizard. Please don't alter the install location. By clicking *Finish* the *zen Developer* plugin is loaded from the *zeos* website (or your local hard disc) and installed into the following folders below your Eclipse installation:

- eclipse/features
- eclipse/plugins

Please finish the installation by restarting Eclipse.

License key

A newly installed plugin is not activated by Eclipse 3 by default. Please select *zen Developer* from the menu *Window->Open Perspective->Other* to do so.

If this is your first time installation, a dialog will remind you to install the *zen Platform license key*.

Please copy the license file you received from us into the following directory below your Eclipse installation:

plugins/de.zeos.zenit.lib.runtime_2.x.y

just like mentioned on the Welcome-Page. x and y stand for the highest currently installed version of *zen Developer.*

If you haven't received a license file so far, you can obtain one from:

http://www.zeos.de/support/download/license.php

After these steps, Eclipse and the *zen Developer* plugin are installed and operational. You can find the release notes in the directory *plugins/de.zeos.zenit.e3_2.x.y* of your Eclipse installation.

3 Preparations

The *zen Perspective* is now activated. The currently active perspective is indicated on the *shortcut bar* at the left edge of the Eclipse screen (see Figure 2).

The shortcut bar of Eclipse 3 is located at the right top area of Eclipse 3. By clicking on the shortcut bar with the right mouse button and selecting *Dock On -> Left* you can move it to the left edge.

The name of the currently active perspective is displayed in the window title. If you move your mouse over the icons, the names are displayed as tool tip. You can also use the menu *Window->Open Perspective* for switching to defined perspectives.

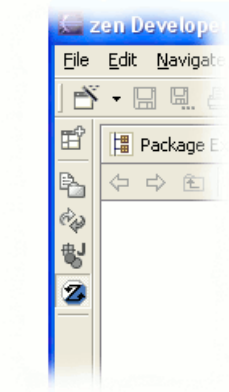

Figure 2
Perspective icons on the shortcut bar on the left edge; the zen Perspective is active.

> The layout and position of Eclipse windows (views and editors) that belong to a logical group are called perspectives. If you change the position of views or editors or close/disable views, you can always return to the preconfigured layout by selecting *Window->Reset Perspective.*

zen Perspective

Modeling a *zen Application* is always done using the *zen Perspective*. Eclipse with an active *zen Perspective* is shown in Figure 3.

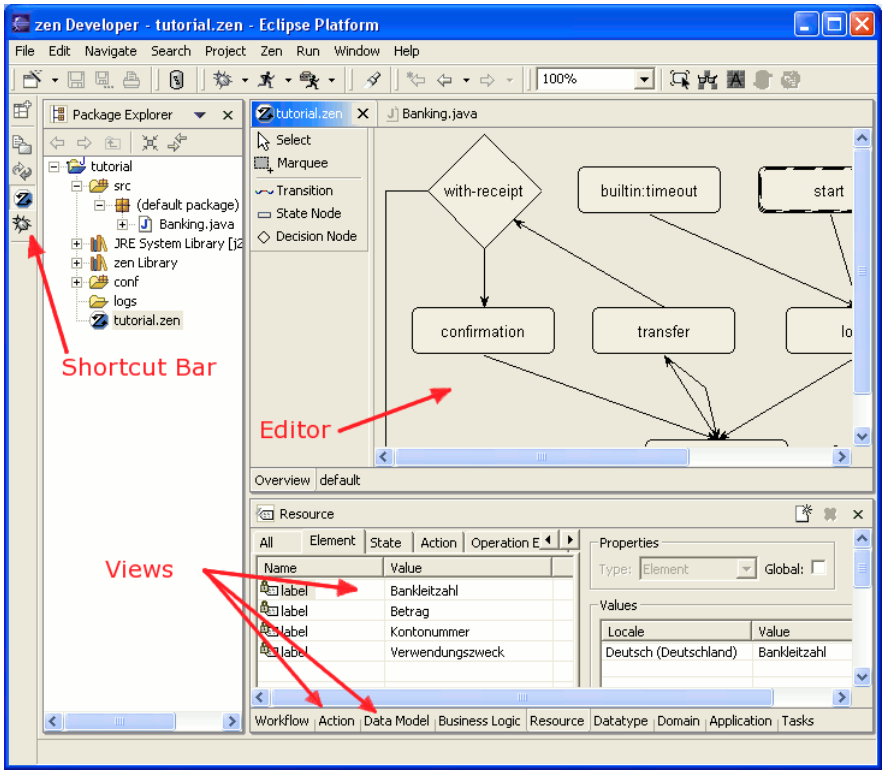

Figure 3
Overview zen Developer

On the left hand side you see the shortcut bar where you can switch between perspectives. Next to the right you see the *Package Explorer* used for navigation within your projects. This screen shot shows only one open project called “tutorial”. On the right side you see the editor on the upper part currently showing the file “*tutorial.zen*” and another tab representing another editor for editing the file “*banking.java*”. Below the editors all views of the *zen Developer* are grouped together. The currently active view is the *Resource View*. Other views can be activated by switching the respective tabs of the views.

e3	Eclipse 3 displays these view tabs on top of the views. You may change this using the *Workbench->Appearance section of the Window->Preferences* dialog.

Creating a zen project for the tutorial

There is no special project type for a *zen Application*. Using a standard Java project is sufficient as the business code is developed in Java as well.

- Create a new Java project by choosing *File->New Project...*
- Choose *Java Project* as type and name it `tutorial`.
- *Optional:* If you want a professional structure we recommend a separate directory containing the source files. To do so click *Next* and click the *Add* button on the *Source* tab. Now use *Create New Folder...* to create a directory called *src*. Confirm the dialog suggesting a directory *bin* as output path for the compiled class files (result is shown in Figure 4).

e3	Eclipse 3 adds the possibility to use such a layout on the second page of the new project wizard without the need to manually add a new source folder.

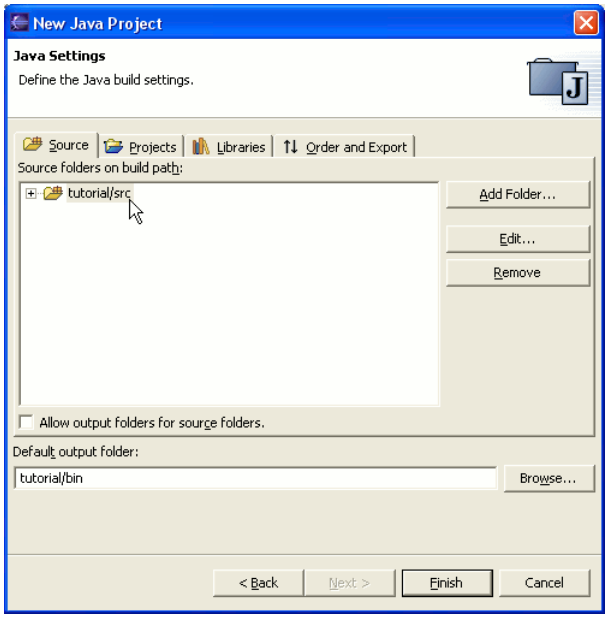

Figure 4
Complete dialog: New Java Project

- After clicking *Finish,* Eclipse suggests changing to the *Java Perspective.* Please choose *No* here to stay in the *zen Perspective* for now.

This project you just created is used to for creating a *zen Application* explained below in detail.

Eclipse workspace

Eclipse manages all resources within a so called *workspace*. It automatically creates a directory named *workspace*. If you did not specify a special location this directory is located under

- *Windows*: in the installation directory of Eclipse
- *Linux/UNIX*: in the *current working directory*, the directory, where Eclipse has been started

The newly created project *tutorial* is located below this workspace structure as well as some of the files that have been installed together with the *zen Developer*:

- *zen-banking:* A completed version of the tutorial project. You can explore and try it by importing it using the menu *File->Import...->Existing Project into Workspace*
- *zen-db:* The integrated *hsqldb* database is preconfigured to store its files in this directory. This database is used to as repository for the application model as well user data when implementing the business code. It is running in background an needs no further attention.

4 Creation of first running application

Within the project just prepared before, the online banking application will be created and enhanced step by step.

4.1 Creation of a zen Application

Please make sure that you have activated the *zen Perspective* or do so now by clicking on its icon in the shorcut bar. Now you can create a new *zen Application* with the aid of a wizard.

- Select the project *tutorial*.
- Choose *File -> New* and the type *zen Application* from the menu. Now the *zen Application Wizard* is displayed. The project *tutorial* is already filled in.

If you are about to create the first *zen Application* within a Java project, you need to prepare a *zen Configuration* first.

> A *zen Application* needs several configuration files. These files must be present prior to opening, editing or executing a *zen Application*. For further details about this configuration please refer to the *zen Platform Developers Guide*.

- Create an initial *zen Configuration* by clicking the button *Create*.
- Change the filename to `tutorial.zen` and choose `zen.repository` as repository for your application and name this application `tutorial` (result is shown in Figure 5)

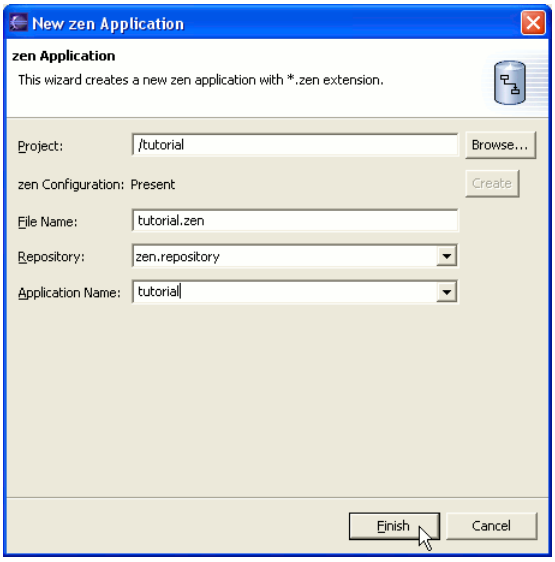

Figure 5
Completed dialog: New zen Application

> The *repository* is a database where all the model information of a *zen Application* is stored. The entry *zen.repository* refers to the preconfigured file-based *hsqldb*. The necessary steps to use a different database are explained in the *zen Platform Developers Guide*.

- Choose *Finish*.

An empty *zen Application* is created and opened automatically. Now that all preparations are completed, you can start modeling the tutorial application. First the data model is defined in a few steps, followed by the definition of the workflow.

> You may pause the tutorial at any time. Save your current work by choosing the menu *File->Save* just like in any usual application. You can reopen a previously closed *zen Application* by double-clicking or using the context menu on the corresponding file (in this case the file named *tutorial.zen*). You may continue your work just were you left.

4.2 Definition of the data model

Considerations

In the first step, the data model will be defined. We will figure out what kind of data we may need in a banking application:

- account information, like account number and balance
- account holder information, like name and address
- transfer including amount, date and note to recipient

In our example we'll start by defining an account consisting of account number and balance. The data model will be completed step by step afterwards.

The data model of a *zen Application* is hierarchical and therefore displayed in a tree structure. It consists of all data needed for input, output or as arguments to operations within the application.
Starting with the fixed root element *data* the model is defined by basic data structures like atom, composition and list.

- **Atom**: an atom is the leaf within an hierarchical data model. Atoms have an assigned data type and contain values. In a web environment, atoms represent input and output fields.
- **Composition**: Compositions do not contain values themselves, they are used to group different elements together. Compositions may contain atoms, lists or compositions.
- **List**: A list is defined on a composition or an atom. During runtime it may contain an arbitrary amount of elements of the defined type.

Using these elements you can define an easy, well structured and very flexible data model.

Basic objects are defined as atoms. They may be grouped together using compositions and lists. The data elements used throughout this tutorial use the shipped data types, so there is no need to define custom data types.

Modeling a *zen Application* is accomplished by using the views of the *zen Perspective*. They are located at the bottom right hand side of Eclipse. Each of them is brought to front by clicking on the tabs or choosing them from the menu *Window -> Show View.*

The data model is defined using the *Data Model View*. Activate it now in order to model the account information.

Account information

The account is created as an composition `account`:

- Select the root element `data`. Open the context menu by clicking the right mouse button. Choosing *Add Composition* creates a new composition which is inserted below the root element.
- Double-click the name of the new composition. The name is now editable. Rename it to `account`.

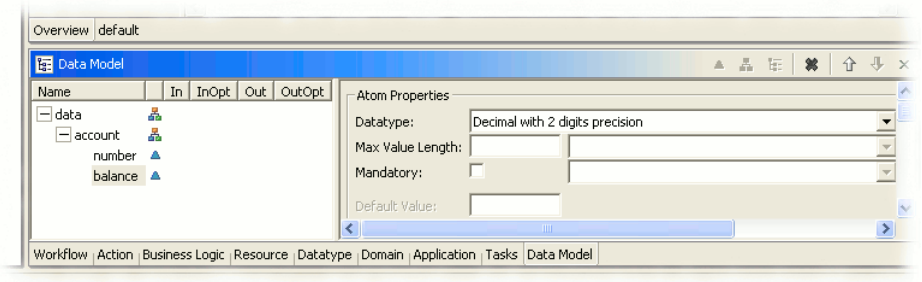

Figure 6
Data Model View: initial data model

- Now select the composition `account` and create a new atom using the context menu. Name it `number` and choose the data type `Long` from drop down box on the right side of the view.
- Add another atom to the composition `account` in order to store the current balance. Name this atom `balance` and assign the data type `Decimal with 2 digits precision` to it.

Figure 6 shows the resulting data model. It is the basic structure used in the following steps when the workflow is added to the banking application.

4.3 Workflow definition

Now a simple workflow is added consisting of two steps: The user should enter his account number to log into the application. To keep the example simple we omit any kind of authorization here. The following page will just display the account number for now.

The worklfow is essential to the behavior of a *zen Application*. It consists of **state nodes**, i.e HTML-pages or other input/output interfaces. These state nodes are connected by **transitions**. The transition from one state node to another is activated by an **action** which is assigned to this transition.

The workflow definition is done in a visual manner using the *zen Editor*. The properties of the workflow elements are edited using the *Workflow View*. Please activate it now.

State Nodes

➢ Draw a state node by clicking on the state node tool in the *zen Editor* toolbox and then click and drag a rectangular shape anywhere on the empty editor canvas (see Figure 7).

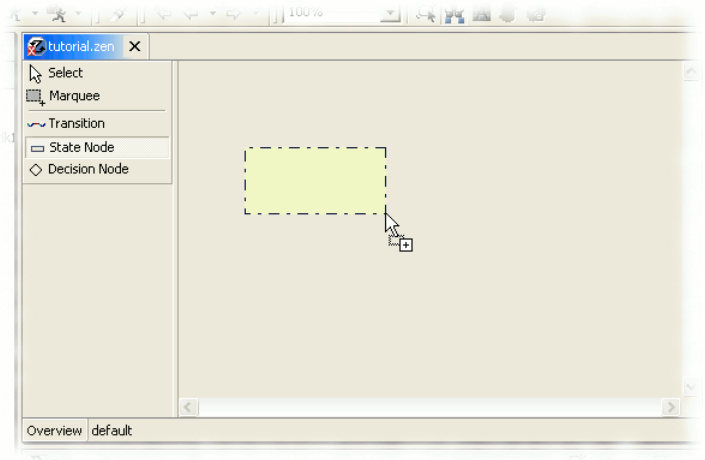

Figure 7
Creation of a state node

After drawing the node, the *zen Editor* resets the state node tool automatically to the *Select* mode.

➢ Using the *Workflow View,* rename the state node to `start`. Confirm your entry by hitting the return key. Choose `defaultentry` as gate from the drop down box in the *Workflow View* (result shown in Figure 8).

The gate property determines whether a workflow may be entered or left from a specific state node.

- *default*: normal state node within a workflow
- *defaultentry*: state node is the central entry point into a workflow
- *entry*: (additional) entry point into a workflow
- *exit*: exit of a workflow

Additional information can be found in the *zen Platform Developers Guide.*

The new state node is currently marked with an error as there is no attached transition originating from it.

This state node is now the entry into the workflow. The next state node will provide the ability for the user to enter his account number.

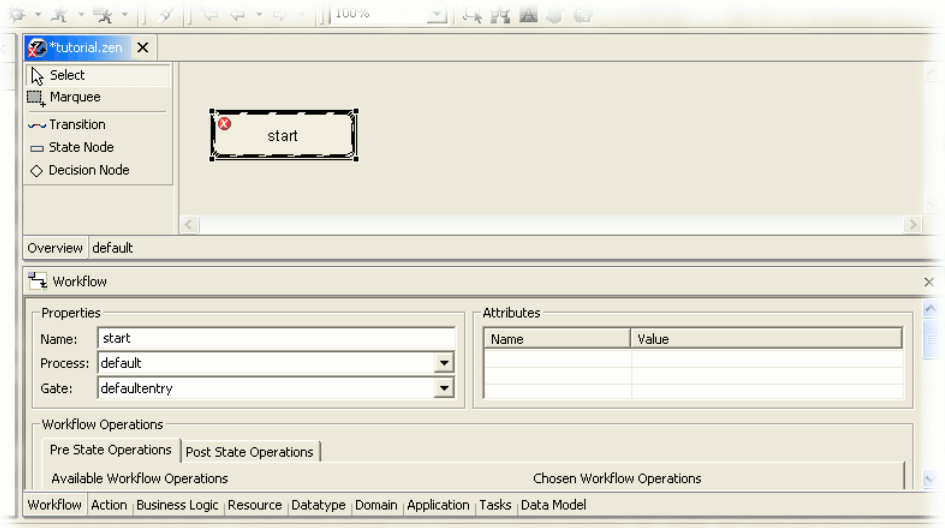

Figure 8
Workflow View and zen Editor showing the first state node

- Draw a new state node and name it `login`. Don't alter the gate property this time and keep `default`.
- Draw a third state node named `menu`, again without the need to change any of the defaults (result shown in Figure 9).

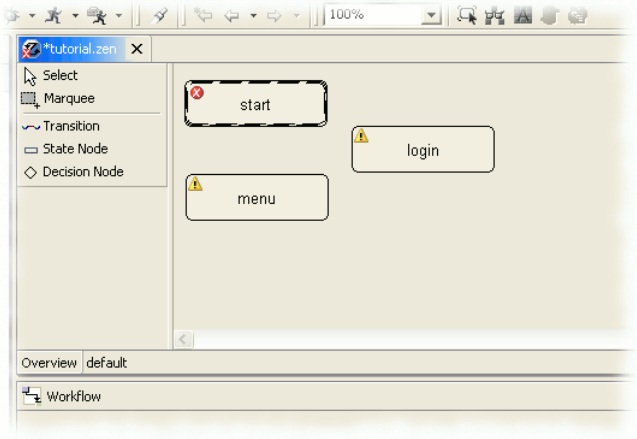

Figure 9
zen Editor showing the first three state nodes

Transitions

The transitions from one node to another node are done using the *Transition* tool of the *zen Editor*.

- Click on the *Transition* tool, notice the symbolic plug-shape of the mouse cursor.
- First click on the node `start` to determine the starting point of the transition. Then click on the state node `login`, thus defining a transition from `start` to `login` that may be triggered by the user.
- In the same way create a transition from `login` to the state node `menu` (result shown in Figure 10).

To delete any workflow element, first select it then delete it either using the context menu, the *Del* key or the *Edit->Delete* menu. Each operation may be undone or restored using the usual Undo/Redo menu.

After the state nodes have been connected, all warning and error signs disappear automatically.

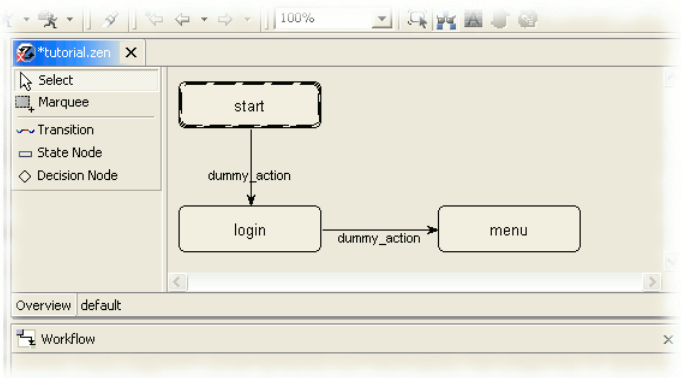

Figure 10
Workflow with transitions

4.4 Connecting workflow and data model

You have just designed a data model and a workflow. Now data model and workflow are put together in order to define what data is displayed or accepted as input on the state nodes.

- Reset the *zen Editor* to the *Select* mode by clicking on the *Select* tool.

The login page should present the account number as an input field. The account number is therefore going to be defined as *In* as well as *Out*. To do so please activate the *Data Model View*.

- Select the state node `login` within the *zen Editor.* Now you can define the input and output elements of this state node.

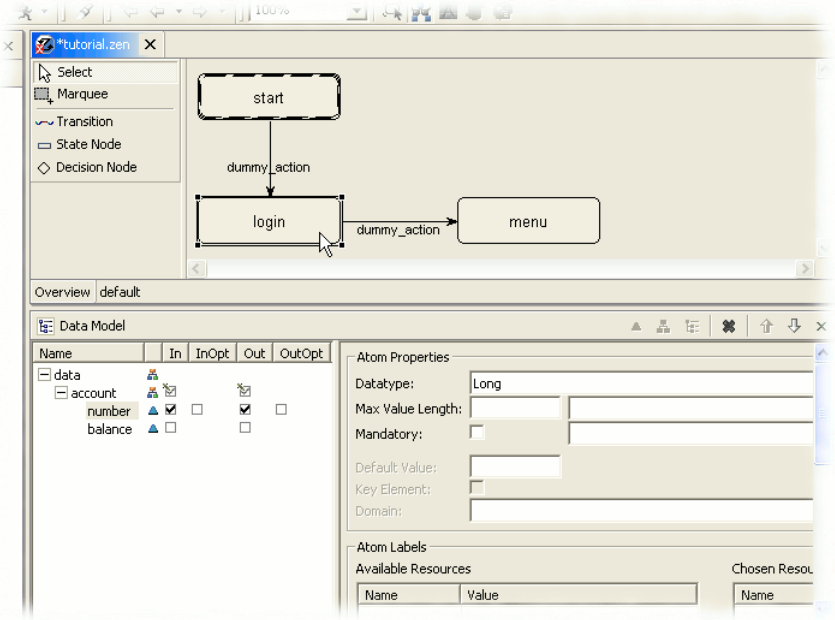

Figure 11
In *and* Out *properties assigned to the state node „login“*

- Using the *Data Model View,* first click on the element `number` then tick the check boxes of this row in the columns *In* and *Out* like in Figure 11.

The account number entered by the user should be displayed on the page `menu` but it is not an input field there.

- Now click on the state node `menu` and then select the the element `number`. This time, only tick the check box *Out* (result shown in Figure 12).

- Save your model by choosing the menu *File->Save.*

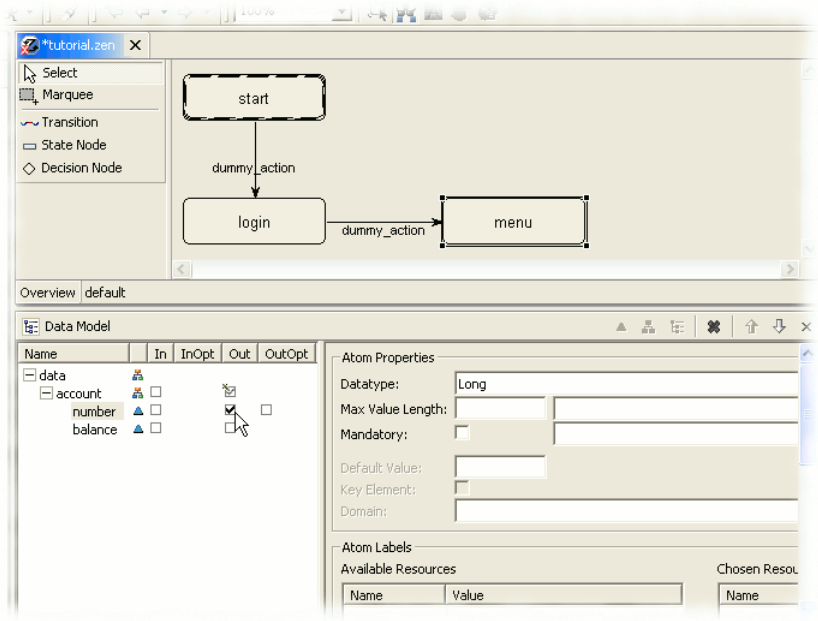

Figure 12
Out *elements assigned to the state node „menu“*

Now everything is in place for a first launch of the application. Eventually Eclipse activates the *Task View* after saving the model; you may ignore this for now.

4.5 Development in different languages

The *zen Platform* supports the development of multi-language applications. All resources necessary for display, e.g. texts and image paths, can be defined in different locales. A locale is the combination of a language and a country, therefore *English (United Kingdom)* and *English (United States)* are considered as two different locales. When the application is being executed in a given locale, all resources will be displayed in this locale automatically. The provided datatypes of atoms consider the runtime locale as well, so users can input their data in the runtime locale.

To support development and execution of multi-language applications, the *zen Developer* uses two different locale settings:

- *Default Runtime Locale*: This setting determines the locale the application will use as default at runtime. The *Default Runtime Locale* is set in the *Application View*. For running in other defined localizations, the application has to be called with special parameters. Additional informations on this topic can be found in the *zen Platform Developers Guide.*
- *Developer Locale*: This setting simplifies the entry of locale-specific resources and data during modeling. It is selected with the drop down box at the top of the *zen Developer*. This setting has no impact on the runtime of the application.

For correct execution of an application, all used resources have to be present in the desired locale. The *zen Developer* is shipped with several predefined resources in *German (Germany)* and *English (United States)*. If your computer uses a different setting we recommend the usage of one of these two locales. Before you continue please check the following:

- Activate the Application View and make sure the *Default Runtime Locale* is either set to *German (Germany)* or *English (United States)*
- Select the same locale you set as *Default Runtime Locale* as *Developer Locale* from the drop down box in the tool bar at the top of the *zen Developer* (result is shown in Figure 13).

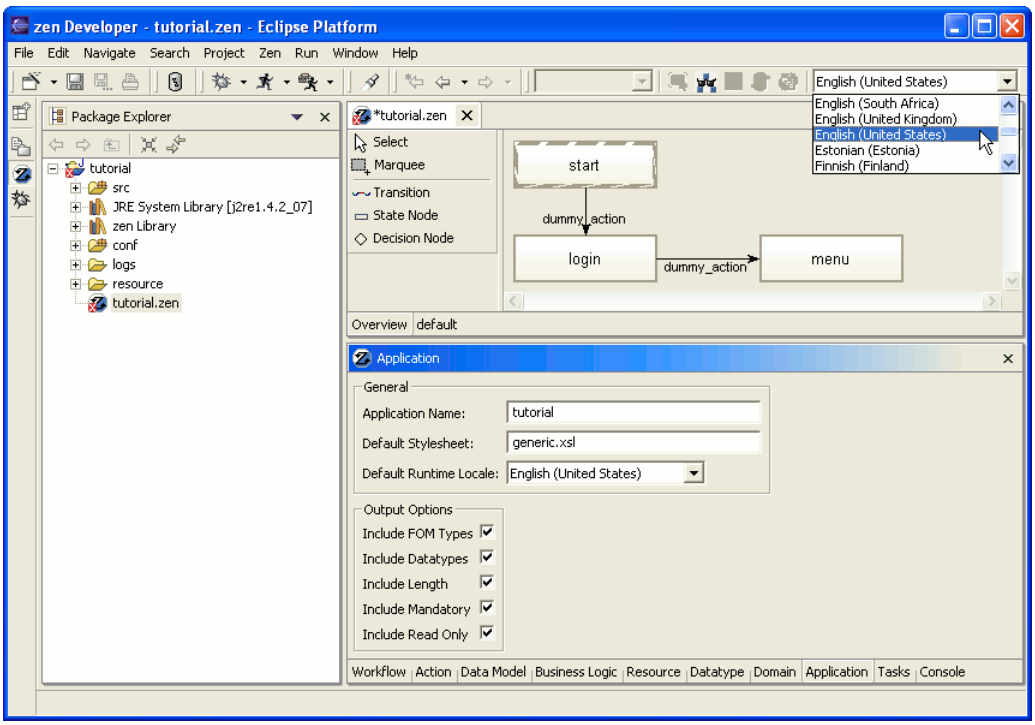

Figure 13
zen Developer: Default Runtime Locale and Developer Locale (drop down at top) should be set to English (Untited States)

If you changed the *Default Runtime Locale,* save the application again.

4.6 Launch and test of the model

The model is ready for execution. Just define a new so-called *debug launch* in Eclipse.

Debug launch

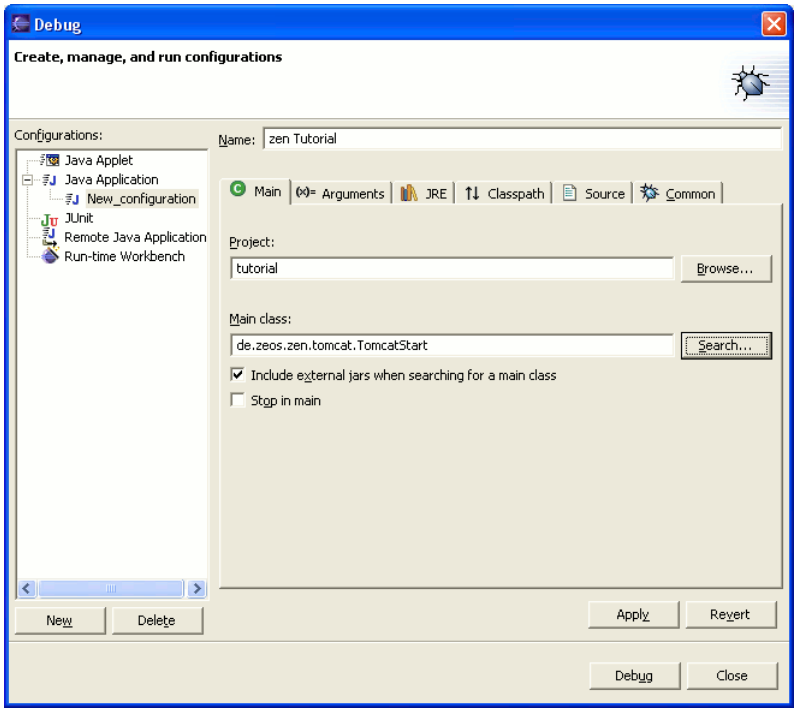

Figure 14
Completed dialog: Debug Launch

➢ Activate the menu *Run -> Debug...* Select *Java Application* and click on *New* in this dialog.

- Name this configuration `zen Tutorial` and select `tutorial` as project.
- Select the option *Include external jars when searching for a main class* and click on the button *Search...* next to the input field. Select the class `TomcatStart`, located in *zenapi.jar* (result is shown in Figure 14).

Start of the zen Application

- Now start the *zen Application* by clicking on *Debug*.

A dialog pops up, warning you that the application has some errors or warnings. The application still lacks error handling which is not needed for our initial start and will be added later. Just acknowledge the dialog for now.

Eclipse automatically changes to the *Debug Perspective*. Wait a few seconds for the debug process to start. If the line

```
[TomcatStart]  Tomcat is up & running.
```

is displayed in the *Console View* the startup was successfull (see Figure 15).

> e3 Eclipse 3 does not automatically change to the Debug Perspective. Only if a breakpoint is hit, a dialog will ask if it should be activated now. Therefore you have to manually change to the Debug Perspective.

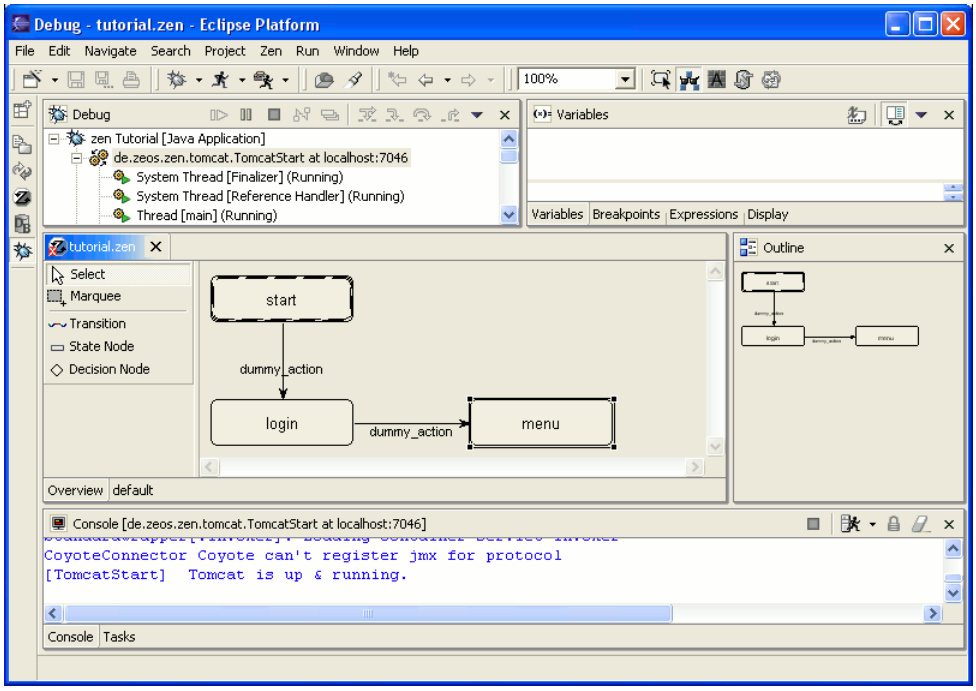

Figure 15
Debug Perspective with running zen Engine (using Tomcat)

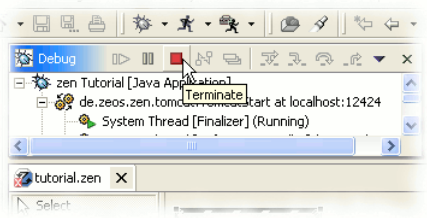

Figure 16
Debug View: Termination of the debug process

The *zen Engine* is running until explicitly stopped, or the entire Eclipse is closed. To tell if it is running take a look at the *Debug View* at the top left area of the *Debug Perspective*: If there are still active threads (not marked as *terminated*) it is still running (see Figure 16). Changes to the model are automatically reloaded by the *zen Engine*, so there is no need to restart it after application modifications.

To stop the *zen Engine*, click on the red button *Terminate* of the *Debug View* (see Figure 16).

- In order to test your application, right-click on the state node `start` and select *Launch Browser from „start“....*This will start up your web browser displaying the first page of your *zen Application*.

If you want to manually launch an external browser you may use the following URL:

`http://localhost:8989/zen/tutorial/zen.repository`

Depending on your operating system, Eclipse may hide your browser by bringing itself into foreground. Please check your taskbar of no browser window is visible.

Testing the zen Application

In your browser you can see the input field for the account number. On the top there is a submit button labeled *dummy_action* (see Figure 17).

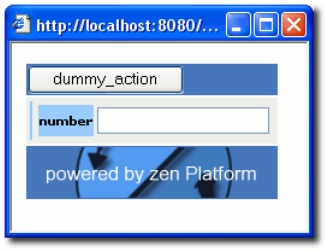

Figure 17
Browser: state node „login“

The HTML output is generated automatically using xsl-transformation. There is no need to design HTML-pages manually. The default xsl stylesheet used in these transformations – named generic.xsl – displays the data using the information of the data model. The used stylesheet can be selected using the *Application View* and may be exchanged by a custom stylesheet easily.

- Now type “Hello” into the input field and submit the form using the submit button.

The input field is rendered with a red background, and an error message „*Wrong format*“ is displayed. This happens because the field *number* has been defined with the data type *Long* and is therefore only accepting digits as input.

The error messages of data type violations are customizable. The data types are defined in the the *Datatype View,* error messages are defined via the *Resource View.*

- Now enter a number and submit the form again.

The application switches to the state node *menu* and the number you have just entered is displayed as simple text output.

- Now close the browser window.

4.7 Changes at runtime

All changes to the application model may be performed at runtime. However, after saving the model a new browser instance should be started.

Now some properties are going to be changed and reflected into the running application.

Mandatory fields

- Open a new browser window using the context menu of the state node *start* again. Leave the input field empty and submit the form.

The application switches again to the state node *menu*, displaying an empty *number* field. This should be prevented by making the account number a mandatory field.

- Close the browser window.

Change the perspective from the *Debug Perspective* back to the *zen Perspective* and activate the *Data Model View*.

- Click on the element `number` and tick the check box `mandatory` on the right side of the view.

- The dropdown box next to the check box is now active and you can select `Missing required entry`.

This text is displayed as error whenever the user misses to fill in this field.

Applying changes at runtime

➢ Save the model now.

If the *zen Engine* is still running, a dialog will pop up requesting whether you want to apply the changes to the running application. Otherwise start the *zen Engine* again using the *Debug-Launch*.

➢ Answer „*Yes*“ to the question and start a new browser window using the context menu of the state node *start*.

The description text of a mandatory field will be automatically rendered with an asterisk (*) by the stylesheet *generic.xsl*. If you try to submit the form now without filling in the account number, the application will remain on the same state node, the field will be displayed with a red background and the error text *“Missing required entry”* will be displayed.

5 Enhance the application

The following enhancements may be performed while the application is still running. Just confirm the dialog to apply the changes after saving.

If you stop the *zen Engine* in the mean time, you must restart it again using the Debug-Launch *zen Tutorial* you have configured previously.

5.1 Adding state resources

Using a descriptive headline for each page greatly improves the usability of the application. In order to do this, a headline for each state node will be defined. The headlines will be displayed on top of each HTML-page.

Change back to the *zen Perspective* and activate the *Resource View.* This is the place where texts or image links are defined used later on to label workflow- or data-elements.

➢ Click on the tab *State* to create a resource used by state nodes.

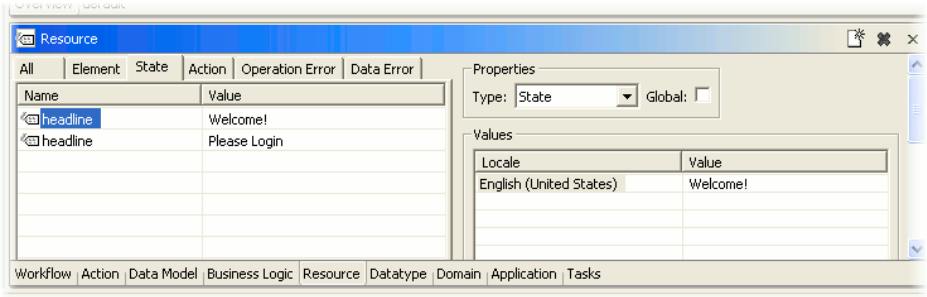

Figure 18
Resource View: labels for state nodes

➢ Right-click into the (empty) table and select *Create* from the context menu

➢ Fill in `Please login` into the column *value* of the new resource `headline`.

➢ Add another *headline*, this time using the text `Welcome!` (result shown in Figure 18).

> The generic stylesheet *generic.xsl* uses state resources with the name *headline* to render a headline onto the pages. For a more complex layout, custom (state) resources may be defined that an adapted stylesheet may use for texts, images or the like.

Activate the *Workflow View* again to assign these new headlines to the according state nodes.

➢ Select the state node `login` in the *zen Editor.*

➢ Select `Please login` from the list of *Available Resources* of in the *Workflow View* and add it to the list of used resources by double-clicking it or using the arrow button (result shown in Figure 19). You may need to scroll the *Workflow* View down to see all resources.

➢ Assign the headline `Welcome!` to the state node `menu` in the same way.

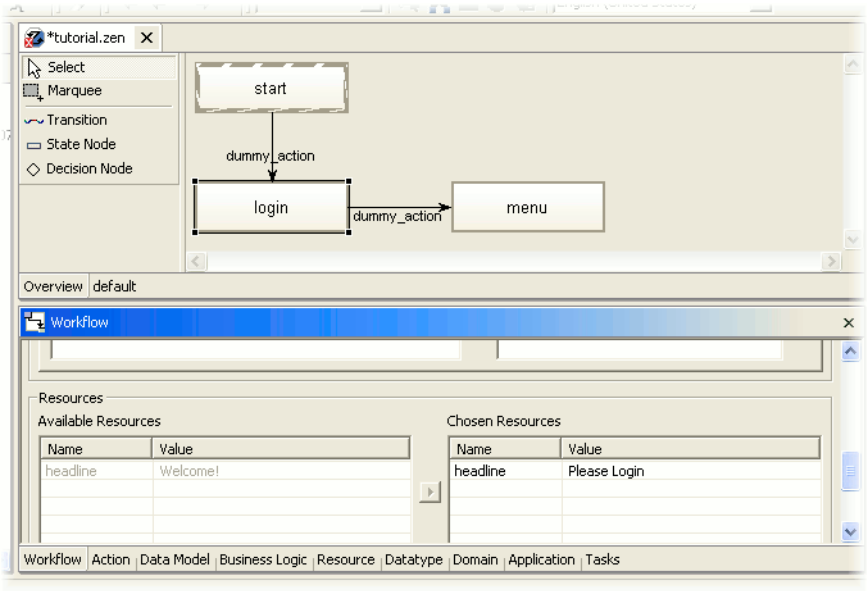

Figure 19
Workflow View: label assigned to the state node „login“

After saving the model, the headlines will be displayed automatically in the application.

5.2 Extending the workflow

Now the workflow will be extended with another step, providing the user with the ability to enter a transfer. Make sure the *zen Perspective* is active, if not change to it.

- Draw a new state node in the *zen Editor* and name it `transfer` using the *Workflow View*.
- Draw a transition from the state node `menu` to the new state node `transfer`. With this state node, the user is enabled to enter all the transfer details later.

Actions

In order to manage the transitions and to define specialized behavior on them, an *action* is assigned to each transition. Actions have to be defined separately. Activate the *Action View* now (see Figure 20). Notice the action *dummy_action*, which has been assigned to each transition automatically so far.

> A possible transition from one to state node to another is defined by a **transition**. The actual switch from one state node to the next is triggered by an **action**. In a web environment this may be i.e. the click of a submit button. Each transition is assigned an action.

You will be creating specialized actions that you may assign to the transitions right now. Using the right mouse button on the action table, you can select *Create* from the context menu. The first action will be used for login.

- Create a new action.
- Change the automatically generated name to `login`.
- In order to create a transfer, create an action `transfer` in the same way.
- To actually execute the transfer create another action named `execute`.
- In order to cancel a transfer, add the action `back` as well. It will be used later on in several transitions. You should set the *Type* to *nonvalidating* for this action only (result shown in Figure 20).

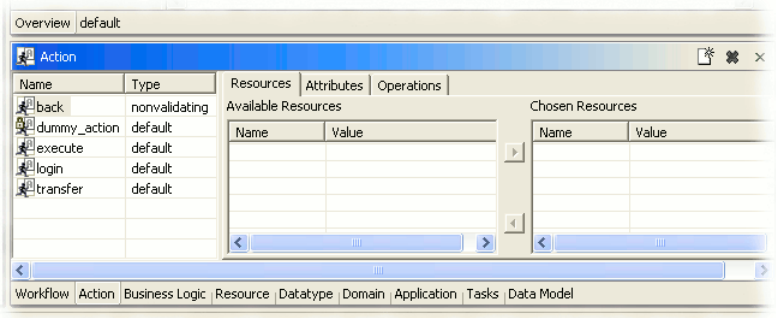

Figure 20
Action View showing some actions

The way transitions work is determined by their *action type*.
The most important ones for now are:

- *default*: All input data is validated.
- *nonvalidating*: No data validation. The current state may be left even if there are user errors on the form. The input will be saved but is not available to operations. Business rules will not be executed and no errors will be displayed.

For further information please refer to the *zen Platform Developers Guide*.

These new actions are now assigned to the appropriate transitions.

- Select the transition between `login` and `menu` using the *Select* tool of the *zen Editor*.
- Using the *Workflow View*, select the action `login` from the action dropdown box instead of the action *dummy_action*.
- Now select the connection between `menu` and `transfer` and choose the action `transfer` (result shown in Figure 21).

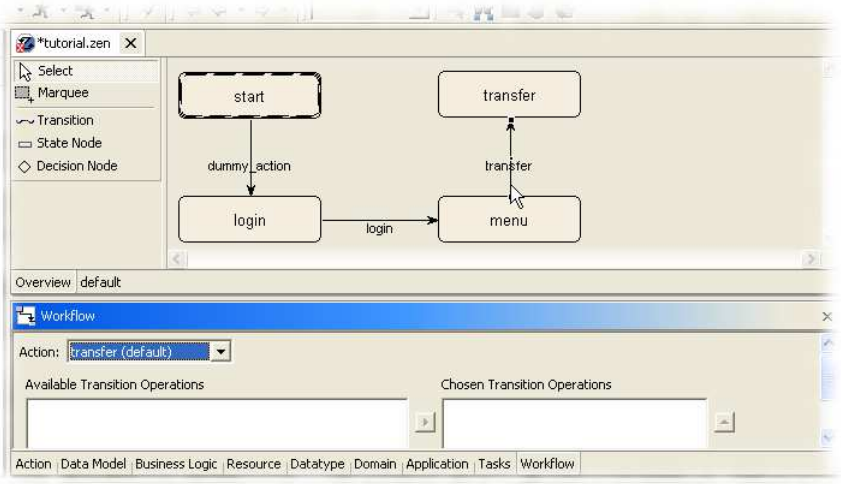

Figure 21
Actions assigned to transitions

After the gathering of the transfer details on the state node *transfer*, the transfer should be executed. After the execution, the state node *menu* should be displayed again, so the user is able to continue his work.

- In order to to do so, create a transition originating from the state node `transfer` ending on the state node `menu` and assign the action `execute` to it.
- To provide the ability to cancel a transfer, create another transition from `transfer` to `menu`. This time assign the action `back` to it (result shown in Figure 22).

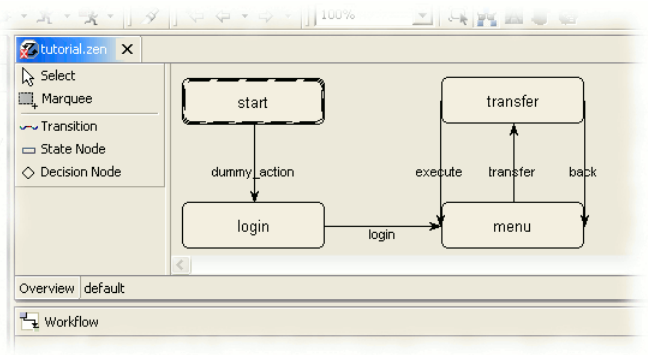

Figure 22
Workflow including the actions „back“ and „execute“

> The function *Auto Layout* provides a convenient way to arrange the elements of the workflow. Choose *Zen -> Auto Layout* from the menu or use the appropriate button of the tool bar. All workflow elements will be rearranged then.

These changes may be executed already, but there is still some work to be done: The details on the state node *transfer* have not been defined yet.

5.3 Extending the data model

To provide the form for entering the transfer details to the user, the data model must be extended. Activate the *Data Model View* now.

- Add a composition named `transfer` to the data element. Refer to Section 4.2 Definition of the data model how to do this.
- Add the following children to this composition:

Name	***Typ***	***Datatype***
`accountno`	Atom	Long
`bankcode`	Atom	Long
`amount`	Atom	Decimal with 2 digits precision
`text`	Atom	String
`date`	Atom	Date
`receipt`	Atom	Boolean

The transfer should be only executed if all fields are filled in. Therefore some of the atoms are defined *mandatory*.

- Click on `accountno` and tick the `mandatory` checkbox on the right hand side of the view.
- The drop down box next to the check box is now active and you can select `Missing required entry` as error message.

This text will be displayed if the user misses to fill in the field.

- Make the atoms `bankcode`, `amount` and `date` mandatory in the same way. Use the same error message as before.

Additional atom properties

In addition you are also able to limit the length of input fields. Enter the desired length restriction into the field *Max Value Length*.

> When using the HTML-output with the *generic.xsl* stylesheet, the length of the input fields are limited to the value of *Max Value Length.* This prohibits the entry of longer values, making a server-side validation redundant. It is useful for other output types and as security measure anyway.

- For instance, set the *Max Value Length* for the `accountno` to `10`, for the `bankcode` to `8` (german bankcodes have a fixed size of 8 digits, feel free to adapt the size to your own needs), and for the `date` element to `10`. Analogous to the mandatory property, set the error message in the dropdown box next to the *Max Value Length* field for each of these atoms to *Maximum length exceeded.*

By adding a *Default Value* you can further define a value for mandatory field that is filled in by default.

- Define the atom `receipt` as `mandatory`, choose `Missing required entry` as error message and enter the string `true` as *Default Value*.

A boolean atom is displayed as a HTML check box by the stylesheet *generic.xsl.* The default value `true` will cause the check box to be checked by default (result shown in Figure 23).

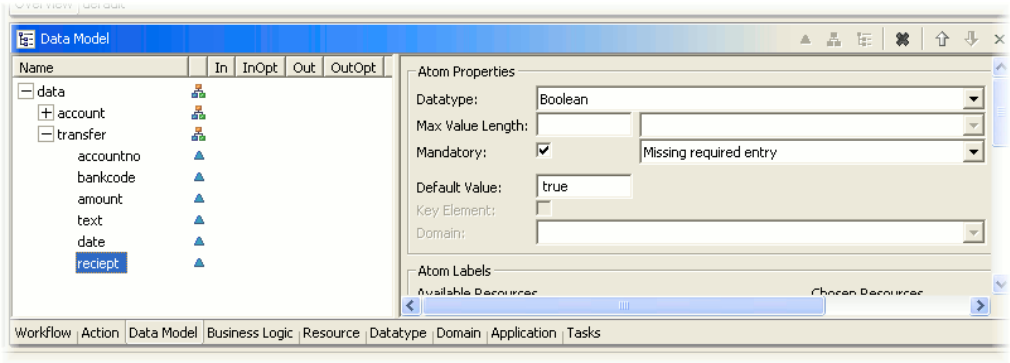

Figure 23
Data Model View: extended data model to be used in transfers

These data elements should now build up the input form for a transfer displayed on the state node *transfer*. In order to do so they must be defined as *in* and *out* for this state node.

- First select the state node `transfer` in the *zen Editor* using the *Select* tool, then click on the composition `transfer` in the *Data Model View.*
- Now tick the check boxes in the columns `In` and `Out` of this row.

By setting the *In* and/or *Out* property on a composition, all children of the composition are automatically set to *In* and/or *Out* respectively.

Test

- Now save the model and test the result.
- Start a new browser by using the context menu on the state node *start.*

In the browser window the new actions are now visible as buttons on top of the screen, as well as the elements and state resources just added (see Figure 24).

- Test the new mandatory property and the validation of date entries on the transfer form as well.

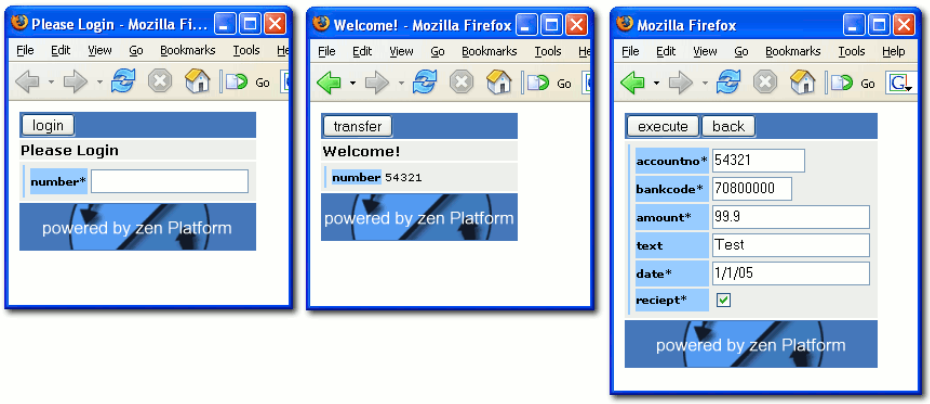

Figure 24
Browser: state nodes „login“, „menu“ and „transfer“

After successful testing change back to the *zen Perspective*.

5.4 Adding element and action resources

Form based applications usually provide a label for each form field to explain what should be filled in. Up to now no special labels have been defined, instead the styleheet *generic.xsl* displayed the names of the elements as hint. To further optimize the output you are able to add specialized, language-specific labels to each entry field or button.

Activate the *Resource View* again. Throughout this chapter language-specific labels will be added to the data model elements and to actions. Resources for elements will be defined using the *Element* tab, Resources for actions will be created using the *Action* tab.

The styleheet generic.xsl will display element and action resources with the name *label* as label for fields or buttons. You just have to enter any text you like to be displayed.

- Select the *Element* tab of the *Resource View*. Define some element labels used by the transfer form, e.g. the values `Account Number`, `Bank Code`, `Amount` and `Purpose`.

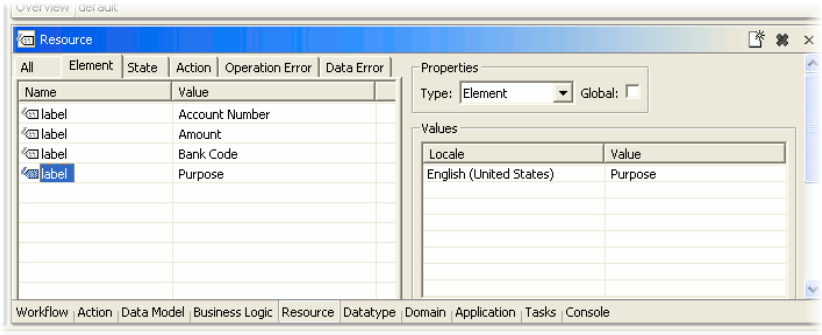

Figure 25
Resource View: element labels

- Activate the *Data Model View* and assign these new labels to the elements `accountno`, `bankcode`, `amount` and `text`. To do so select each of the elements and assign the desired label on the right hand side of the view.

- Now define some action (button) labels using the *Action* tab of the *Resource View*. Enter the values: `Back`, `Execute`, and `Transfer`.

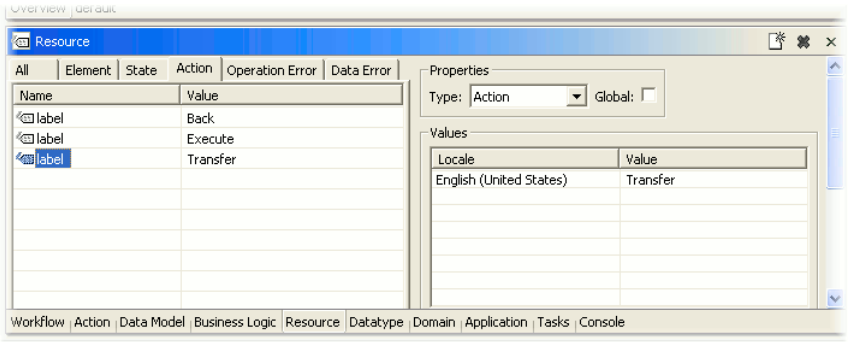

Figure 26
Resource View: action labels

- Now change to the *Action View* and assign the labels to the actions `back`, `execute` and `transfer`.

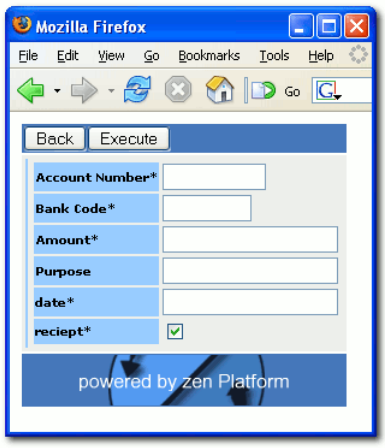

Figure 27
Browser: state node „transfer“ with labels

After saving and applying the changes to the running application, the respective elements and actions are displayed with the labels just defined (see Figure 27).

5.5 Error handling

Triggering a timeout

> If a user session stays idle for some time, all associated data is deleted. This timeout value may be adjusted in the configuration file *scfservice.xml.* It defaults to 750 seconds.
> Further information can be found in the *zen Platform Developers Guide*.

If a timeout or an application error occurs, the *zen Engine* switches to a timeout state or an error state. To test this behavior you may provoke a timeout by doing the following instead of waiting for 750s:

- Start a new browser using the context menu on the state node *start.*

The state node *login* will be displayed.

- Now stop the *zen Engine* using the *Debug Perspective* (see Chapter 4.6 Launch and test of the model). Don't close the browser window! Start the *zen Engine* again by choosing *Run->Debug History->zen Tutorial* from the menu.

This results in loss of all session information, but the browser still refers to its old session.

- Now submit the form by clicking on the login button.

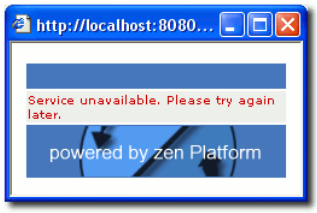

Figure 28
Browser: default general error message

Due to the termination and restart, the *zen Engine* can't find the session requested by the browser. This is exactly what happens if the session times out, thus we simulated this behavior in a quicker way. The browser now displays an error message defined as fall back by the stylesheet generic.xsl (see Figure 28).

Special State Nodes: builtin:error / builtin:timeout

This simple default message is usually not sufficient for a real world application. You can define your own error and timeout states to handle these conditions gracefully. To do so change back to the *zen Perspective*.

- Create two new state nodes in the *zen Editor* and name them `builtin:error` and `builtin:timeout`. Please watch out for the exact spelling here, including upper-/ lowercase..

This resolves the two remaining errors from the *Task View* and the warning dialog on startup of the application will not appear again, because the model is now complete.

> The state nodes *builtin:error* and *builtin:timeout* are activated spontaneous if the *zen Engine* encounters an application error or a timeout. In contrast to ordinary state nodes they don't need any incoming transitions. By using outgoing transitions, however, it is possible to provide a defined workflow after an error occurred.

- Create a transition from each of the new state nodes to the state node `login`. This provides the user with the ability to login in and start over again after an error or timeout happened.
- Assign the action `back` to these transitions using the *Workflow View* (result shown in Figure 29).

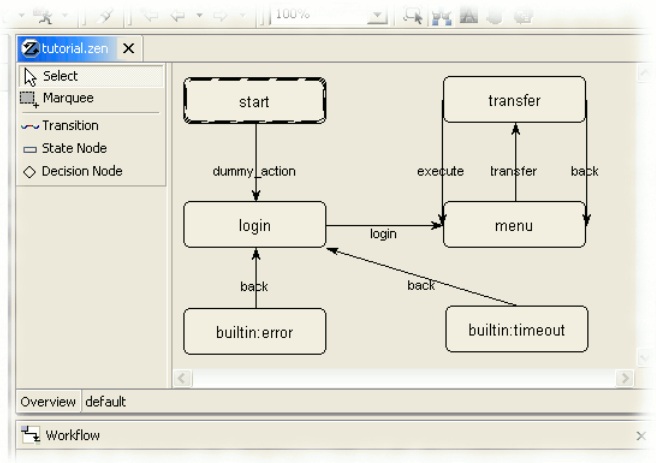

Figure 29
Error handling with state nodes and transitions

- Activate the *Resource View* and create two new state labels on the tab *State*. The headlines should be named `headline` and have the values `An error occurred. Please login again` respectively `A timeout occurred. Please login again`.
- Assign these headlines to the newly created state nodes using the *Workflow View*.
- Save and test the application by provoking a timeout like explained above.

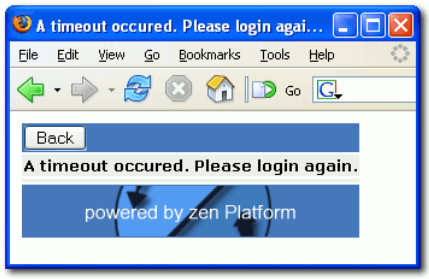

Figure 30
Browser: state node „bultin:timeout“

The *zen Engine* switches to the timeout state just defined. From this page the user can now navigate to the login form (see Figure 30).

> If you want even finer control over the timeout/error behavior, you can create actions named *bulitin:error* and *bulitin:timeout* instead of globally defined error and timeout states. These specialized actions may by assigned to individual transitions providing different behavior depending on the state where the error occurred. For further information please refer to the *zen Platform Developers Guide.*

6 Programing business logic

So far, the *zen Application* was created without a single line of code, but the actual execution of a transfer needs some business code of course.

The business logic of *zen Applications* is coded in Java. Java methods are integrated with the model as so called *operations* and assigned to the data- and workflow model. They are provided with full access to the data model and the service API of the *zen Platform*. For each Java method an operation is defined providing the link between Java and the model.

Eclipse has the ability to change the Java code during runtime of the *zen Engine*. Under some conditions, hot code replacement isn't possible and a warning dialog will remind you that the *Hot Code Replacement* failed. In these cases you must stop and restart the *zen Engine* to test the changes you have made.

To provide the business logic you must create a new Java class at first:

- Click on the project *tutorial* in the *Package Explorer* with the right mouse button.
- Choose *New -> Class* from the context menu. Enter `Banking` as name for this class. The other options can be left by their defaults (result shown in Figure 31).

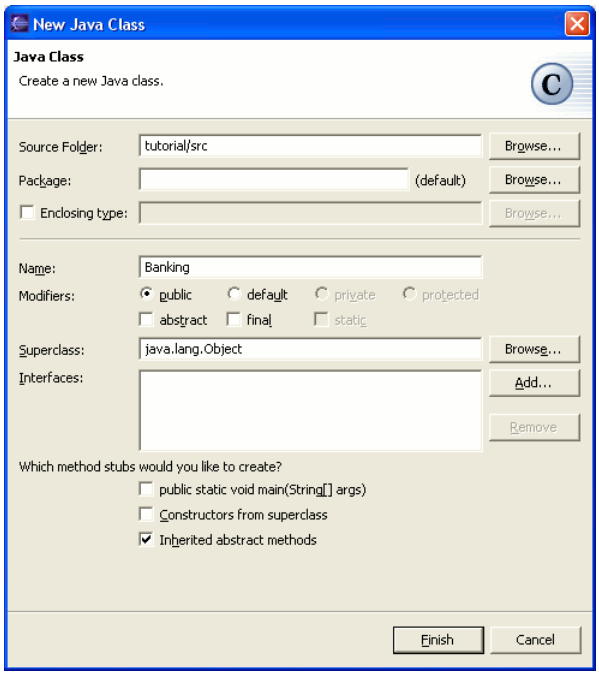

Figure 31
Completed dialog: New Java Class „Banking“

By clicking on *Finish*, Eclipse creates a Java class with your selections and automatically opens the the Java editor displaying it. This is the place where the actual coding is done.

The *zen Platform* has two different categories of operations:

- **Workflow Operations:** Workflow Operations and Decisions are assigned by the developer to individual workflow elements like nodes, transitions, or actions. This determines when they are executed.
- **Business Rules:** Computation Rules and Validation Rules are only linked to the data model. They are executed whenever the assigned data model elements are provided with changed user input.

6.1 Decisions

The workflow of a *zen Application* hasn't necessarily to be static. It may be useful to determine at runtime how to proceed on special conditions. Therefore the *zen Platform* provides *decisions*.

The state nodes of a workflow are connected with transitions. These define all possible state changes in a static manner. A transition may be forked by introducing a **Decision Node**. Based on certain data values, one state is chosen among all possible follow up states.

The banking application is now extended in order to enable an optional receipt for the transfer. The user may tick a check box *receipt* in order to receive an additional summary page. To provide this functionality a decision node is created with an associated decision operation to decide which transition will be followed. An additional state node implementing the receipt page will be added as well.

If you haven't done so, change back to the *zen Perspective* and activate the *zen Editor*.

Extend the model

- Add a state node for displaying the receipt data in the *zen Editor* and name it `confirmation`.
- Choose the decision node tool from the toolbox and draw a decision node onto the canvas.

The decision node is marked with an error until you assign an operation to it. The operation provides the actual decision.

- Name the decision node `with-receipt` using the *Workflow View*.
- Change back to the select mode and locate the transition originating from `transfer` to `menu` that is labeled with the action `execute`.
- To re-route this transition, first select it by clicking in it. Now click and hold the mouse button down on the endpoint and drag it onto the decision node.
- Draw two new transitions from `with-receipt` to `menu` and from `with-receipt` to `confirmation`. These two transitions define the possible branches of the workflow.

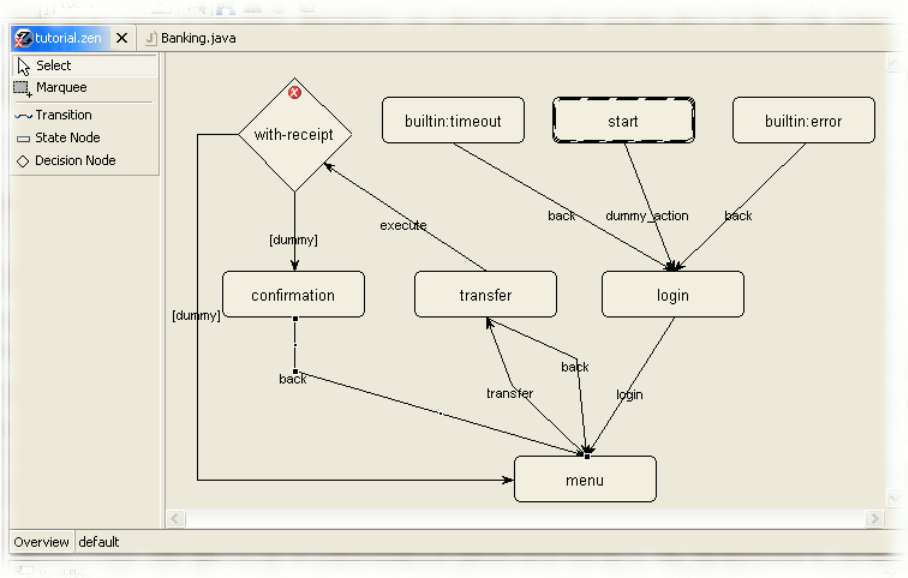

Figure 32
Workflow with decision node

Now the execution of a transfer does not directly switch back to the menu but first routes to the decision node which in turn decides about the proceeding workflow. The decision node may either continue with the state node *menu* or switch to the additional state node *confirmation*.

- Draw another transition from `confirmation` to `menu` and assign the `back` action to it using the *Workflow View* (result shown in Figure 32).

Java method

> A decision operation determines which outgoing transition should be used by the workflow. All required information for this decision may be passed on by the operation parameters.

This decision operation is a simple Java method returning a string to indicate which transition should be taken. In this example it is based on the value of the atom *receipt*.

- Add a method to the Java class `Banking` you have created before:

```
public static String confirm(Boolean b) {
 if (b.booleanValue())
  return "yes";
 else
  return "no";
}
```

Save the changed Java file. Eclipse automatically compiles the source code each time it is saved. There is no need to start the compiler manually.

The next step will link the business code with model. This defines, when the operation will be executed and what parameters are passed to it. To do so activate the *Business Logic View*.

- Click on the tab *Decisions*. Click with the right mouse button in the empty area below and choose *New...* from the context menu.
- Fill in `Receipt needed?` as description and, using the *Browse...* button, choose the class `Banking` you have created as implementing class.

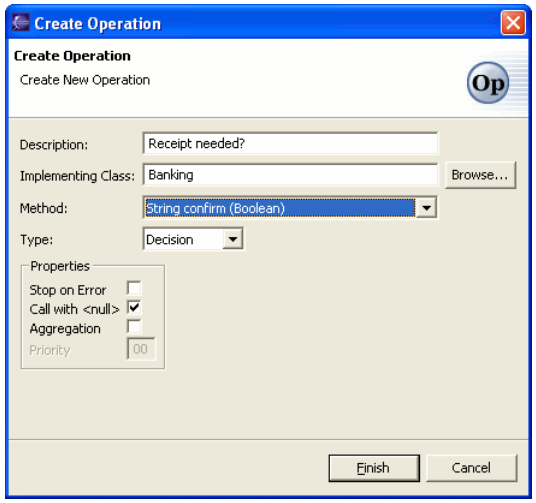

Figure 33
Completed Dialog: Create Operation „confirm“

- From the method dropdown box select the `confirm` method (result shown in Figure 33). Finally click on finish to confirm your creation.

The new operation is marked with an error for now because no arguments have been assigned yet.

This will be fixed right now:

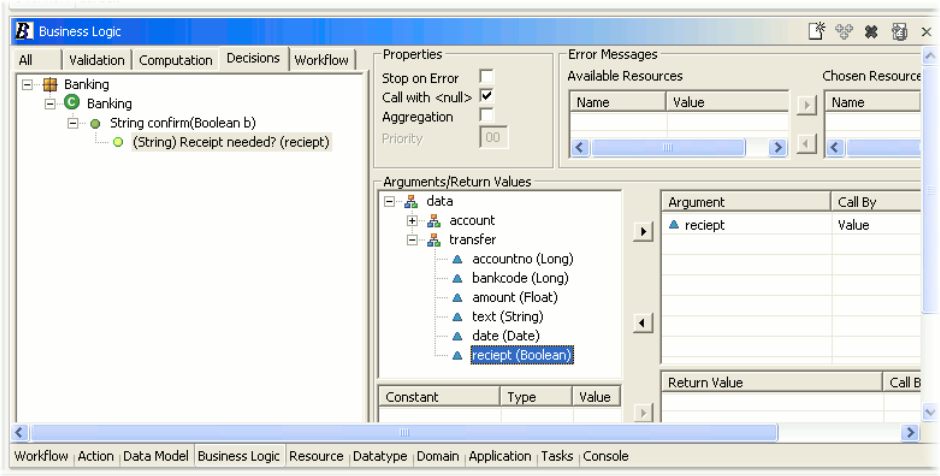

Figure 34
Business Logic View: assignment of arguments to an operation

- Unfold the data element tree shown in the *Arguments* part of the *Business Logic View* until the all children of the composition *transfer* are visible.

- Click on the atom `receipt` and then on the button with the arrow facing right towards the argument table (result shown in Figure 34).

The Java method is now properly connected to the model, the error mark has disappeared. The next step is to associate the operation with the decision node in the workflow.

- Activate the *Workflow View* and select the decision node `with-receipt` in the *zen Editor*.

- Unfold the tree showing the class hierarchy in the *Workflow View* up to the decision operation `Receipt needed?` you just created and click on it (see Figure 35).

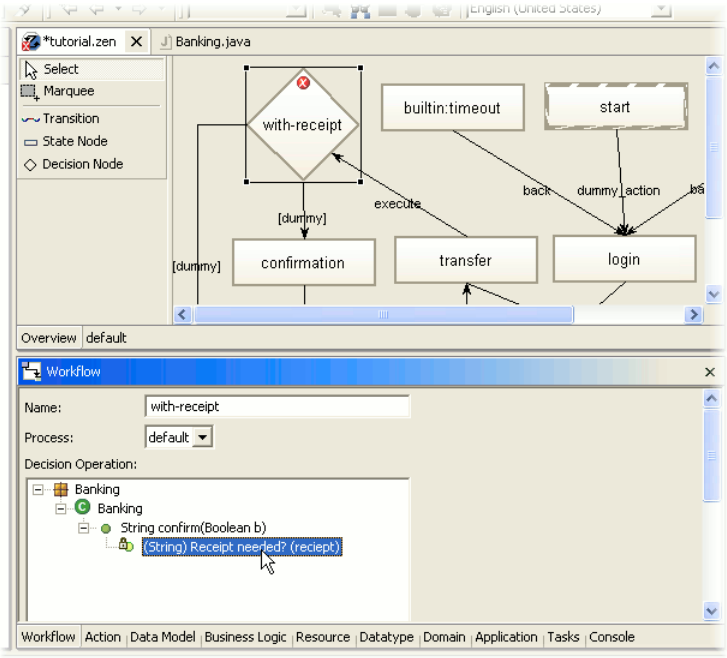

Figure 35
Decision operation associated to decision node

Finally the return values of the decision operation are going to be connected to their respective transitions to determine the desired workflow.

- Click on the transition from `with-receipt` to `menu` and enter `no` as value into the field *Return Value* of the *Workflow View*. Please check that the spelling, including upper-/ lowercase, is exactly as the return value of the Java method (see Figure 36).

- Now click on the transition from `with-receipt` to `confirmation` and enter `yes` as *Return Value*.

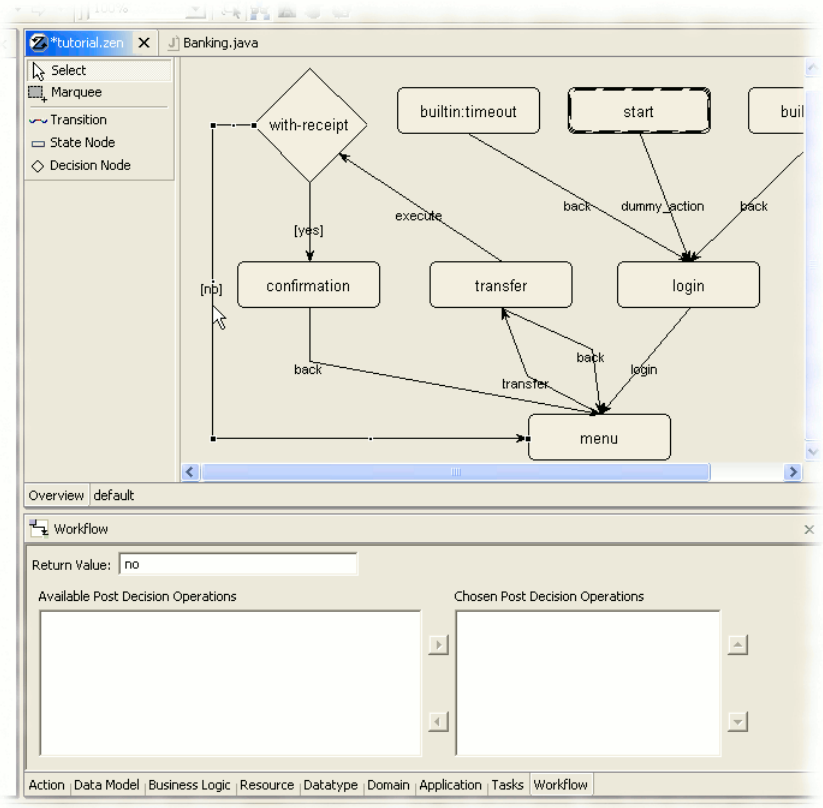

Figure 36
Assignment of the return value to the transition

The error mark on the decision node has disappeared automatically.

If the decision node is now entered, the decision operation just implemented is called with the atom *receipt* as parameter. The decision operation now decides on the following progression of the transition: The transition will be used that has the corresponding return value associated with it.

- The state node *confirmation* still lacks the output of data. To fix this open the *Data Model View,* select the state node and tick the checkbox `Out` for the composition `transfer`.

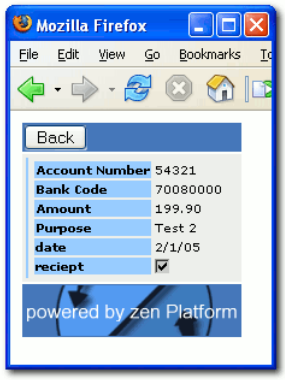

Figure 37
Browser: state node „confirmation“

You now may test the correct display of the receipt (see Figure 37).

6.2 Workflow Operations

In order to execute operations at a certain time during the workflow, so called *Workflow Operations* are available. When the user executes a transfer in the banking application, the transferred amount should be subtracted from the current balance. First some preparations are needed to enable this.

Filling the balance

Missing a complete account management, the current balance is empty. For usage in our tutorial we will put a random amount between 100 and 1000 as default balance.

- Create the following method in the class *Banking*:

```
public static Float generateFloat() {
  return new Float(100.0 + Math.random() * 900.0);
}
```

Save the Java class. An operation is now created for this method using the *Business Logic* View.

- Change to the tab labeled *Workflow* on the *Business Logic View*. Open the context menu by clicking with the right mouse button on the empty area and select *New...*.

- Enter the description `create random balance` and select the method you just created from the `Banking` class. Close the dialog by clicking on *Finish*.

Using the right part of the *Business Logic View,* the arguments will be assigned now. Notice that this method has no input arguments, so only the *Return Value* has to be assigned.

- Assign the atom `balance` (below the composition *account*) as *Return Value* by first clicking on it and then using the lower button that is labeled with a right arrow (facing towards the table of return values). The error mark on the operation should now disappear. Each time the operation is called, the return value is automatically put back into the session data.

The operation should create a random balance each time the user logs in. In order to do so it is added to the transition from *login* to *menu.*

- Activate the *Workflow View* and select the transition from `login` to `menu`.

- Unfold the class hierarchy and assign the operation you just have defined by clicking on the right arrow button. This operation will now be executed along the transition.

- To display the balance, mark the atom `balance` as `Out` on the state node `menu` using the *Data Model View.*

Debit the amount from the balance

- Now the actual method for debiting the transfer can be implemented:

```
public static Float substract(Float balance, Float amount) {
  return new Float(balance.floatValue() - amount.floatValue());
}
```

- Save these changes to the Java class.

- Define this method using the *Business Logic View* by choosing *New...* from the context menu of the workflow operations and describe it as `Debit from balance`. Choose the new method and create it by clicking on *Finish*.

- As input arguments assign the atoms `balance` (below the composition *account*) and `amount` (below the composition *transfer*) by clicking on the arrow button facing right towards the (In)-Arguments. Make sure the order corresponds to the one from the Java method.

- As *Return Value* assign the atom `balance` to the operation as well. Use the arrow button facing right towards the Return Values below.

- Assign this operation to the transition from `transfer` to `with-receipt` using the *Workflow View*.

Now, whenever a transaction is executed, the given amount will be debited from the balance.

6.3 Business Rules

All operations implemented so far have been assigned to specific workflow elements.

Business rules in contrast define a more general rule set that is executed independent of the current workflow. Business rules only depend on the data and are only fired if there is new or changed user input.

With business rules, the *zen Platform* provides operations that are only dependent on data. They are not connected to the workflow by any means. They are executed automatically every time the user enters or changes data associated with them. There are the following types of business rules:

- **Computation Rules:** Business rules that are used to compute or change data. These computed results may trigger additional business rules.
- **Validation Rules:** Business rules that are used to validate data only. They must not change any data.

Computation Rules

As first example we will introduce a computation rule: The text entered as purpose for the transaction should be converted to all uppercase.

- Implement the following Java method:

```
public static String toUpperCase(String s) {
  return s.toUpperCase();
}
```

- Save the Java class.
- Create a new operation in the *Business Logic View* for this method. Click on the tab *Computation* this time and choose *New…* from the context menu of the empty area. Enter `convert to uppercase` as description.
- Assign the atom `text` as both input and output argument to this operation.

This computation rule will be executed by the *zen Engine* automatically each time the user submits a new or changed purpose text.

Test this computation rule by entering a transfer with receipt. The purpose text now shows up in uppercase on the receipt page.

Validation Rules

Validation rules enable you to enforce business-dependent validations. They are executed after (and may be triggered by) computation rules.

The example application does currently not validate if the date of a transfer is in the past. Now a validation rule will be created to make sure that past dates cannot be entered by the user. If he tries to, an error message will be displayed.

- Implement the following method in the class *Banking*:

```
...
import java.util.Date;
import de.zeos.zen.api.core.ValidationRuleException;
...

 public static void checkDate(Date d, Date today) throws ValidationRuleException {
  if (d.before(today))
    throw new ValidationRuleException();
 }
```

The violation of a validation rule is signaled by throwing a *ValidationRuleException*. The user will be presented an error message that is assigned to this operation.

- Activate the *Resource View* and click on the tab *Operation Error*.
- Create a new resource using the context menu. Enter the following text as value: `The date must not be in the past` (result shown in Figure 38).
- Now create a validation rule using the *Business Logic View*: Click on the tab *Validation* and, using the context menu, create a new operation for the method `checkDate` with the description `check transfer date`.
- Assign the element `date` as first argument and the *Field Function* `today` as second argument. Field functions are located at the bottom in the *Arguments/Return Values* part of the *Business Logic View*. Field function are predefined functions that provide some convenience functions like *today* which – as a simple example- returns the current day based on midnight in order to simplify date comparisons.

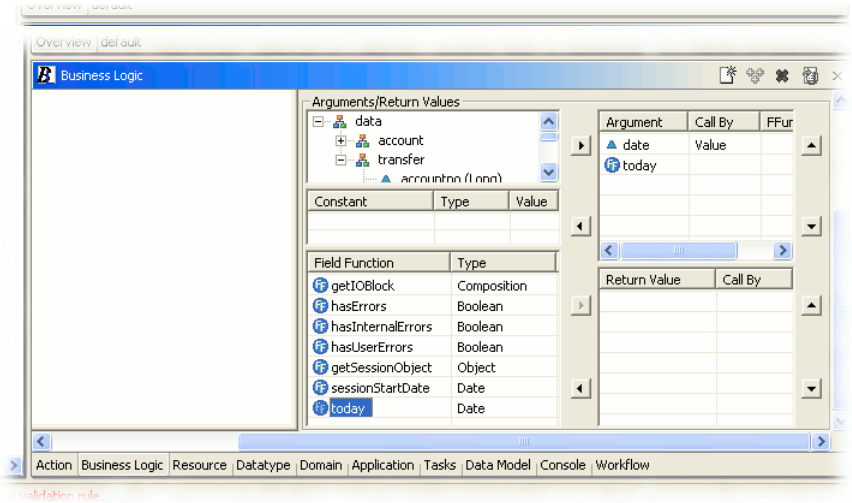

Figure 39
Business Logic View: using a field function as argument

Now the prepared error message must be assigned to the validation rule:

- With the operation *check transfer date* still selected, choose the new error message from the list of *Available Resources* and add it to the *Chosen Resources* via the arrow button.

Similar to the computation rule, there is no need to link this operation to a specific workflow element. It is executed automatically every time its arguments change.

Now if the user enters a date in the past, the modeled error message will be displayed.

If a single validation rule should be able to display multiple error messages depending on different conditions, these error messages must have different names. The *ValidationRuleException* must then be thrown with the desired name in its constructor.

7 Simple persistence via JDBC

Up to now, all application data has been created dynamically and has not been persisted. The following chapter will show how to store the transfers into a database and load them from there as well.

7.1 Store in a database

As a first step, the transfer details the user entered will be stored to a simple database table. The database structure needed for this example has already been defined in the installed *hsqldb* database and can by used right away.

- Add the following code to store data to the *Banking* class:

```
...
import java.sql.Connection;
import java.sql.PreparedStatement;

import de.zeos.scf.service.InitialServiceConnector;
import de.zeos.zen.api.core.OperationRuntimeException;
...

public class Banking {

...

 private static String dbPool = "test.store";
 private static String stmt = "insert into entries " +
  "(account, bcode, text, amount, date) values (?,?,?,?,?)";

...

 public static void book(Long account, Long bcode, Float amount, String text,
                         Date date) {
  Connection con = null;
  PreparedStatement prepStmnt = null;
  try {
   con = new InitialServiceConnector().getConnectionService().getConnection(dbPool);
   prepStmnt = con.prepareStatement(stmt);
   prepStmnt.setLong(1, account.longValue());
   prepStmnt.setLong(2, bcode.longValue());
   prepStmnt.setString(3, text);
   prepStmnt.setFloat(4, amount.floatValue());
   prepStmnt.setDate(5, new java.sql.Date(date.getTime()));
   prepStmnt.execute();
  }
  catch (Exception e) {
   throw new OperationRuntimeException("Insert into 'entries' failed", e);
  }
  finally {
   try {
    if (null != prepStmnt)
     prepStmnt.close();
    if (null != con)
     con.close();
   }
   catch (Exception e) {
    //jdbc error
   }
  }
 }

} // end of class Banking
```

Save the class. The Java code is rather simple and consists mostly of the needed JDBC calls. The access to the database is provided by the *ConnectionService* of the *zen Platform*. The connection pool *test.store* used here is already predefined (in the file *conf/scfservice.xml*).

In case of an error during the database access, an *OperationRuntimeException* is thrown. This will result in transitioning to the *bultin:error* state node.

- Activate the *Business Logic View* and click on the tab *Workflow*.
- Create a new operation using the context menu. Use `book into database` as description and select the class `Banking` with the method `book` you just created.
- Assign the atoms `accountno`, `bankcode`, `amount`, `text` and `date` (below the composition *transfer*) in the given order to the operation as arguments (result shown in Figure 40).

In our example, this operation should be called everytime the user filled in the transfer form and submits it using the action *execute*. Activate the *Action View* to connect the new operation with this action.

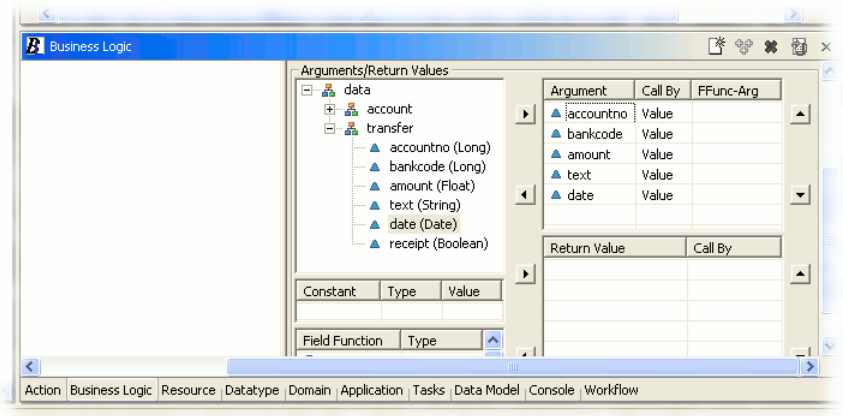

Figure 40
Business Logic View: assignment of the arguments to the operation „book into database“

- Select the action `execute` in the *Action View* and activate the tab *Operations*.
- Unfold the tree so the new operation is visible.

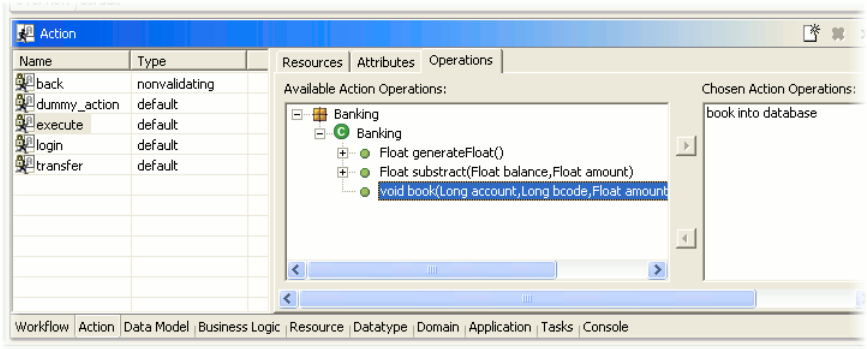

Figure 41
Action View: assignment of the operation to the action „execute“

- Assign it to the action by using the arrow button facing to the right (result shown in Figure 41).

Now every transfer will be stored in the database.

7.2 Load from the database

In order to display previously executed transfers, the application will be extended once more. The transfer details will be loaded from the database and put into the data model to display them. Activate the *Data Mode View* to extend the data model with the necessary elements.

Extend the data model

The transfers will be modeled as a list of single transfers.

- Create a a list named `entries` on the composition `account`.

The list is marked with an error until it is defined what elements the list should consist of.

- Add a new composition `entry` to the list, which itself consists of the atoms `amount` of type `Decimal with 2 digits precision` and `text` of type `String` (result shown in Figure 42).

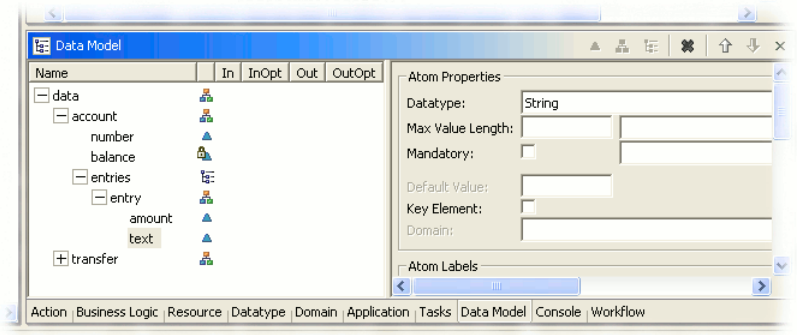

Figure 42
Data Model View: extended data model for persistence

Extend workflow

Now the workflow model will be extended to display the already executed transfers.

- Prepare a new action named `show` using the *Action View*.
- Draw a new state node named `transfers` in the *zen Editor.*
- Connect the state nodes `menu` and `transfers` with a transition and assign the action `show` to the transition.

The user should be able to go back to the menu from the list of transfers.

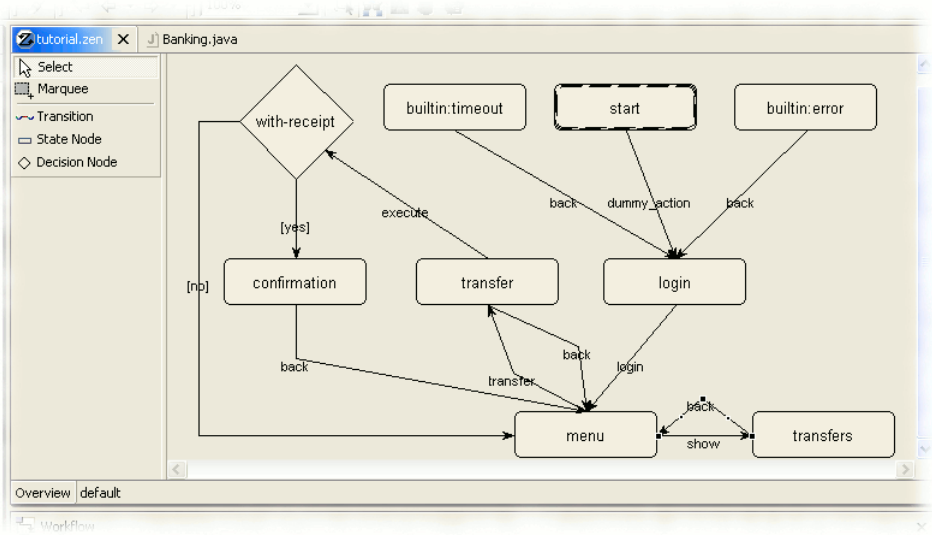

Figure 43
The complete workflow of the tutorial application

- To provide this functionality, draw a transition from `transfers` back to `menu`. This transition should use the `back` action as well (result shown in Figure 43).

The just created state node *transfers* should show all previously executed transfers.

- Change to the *Data Model View* and select the state node `transfer` to define its output data.
- Tick the check box of the `Out` column in the `entries` row (below the compositions *account*).

Due to the fact that the element *entries* is not marked as *In*, the transfers will not be rendered as input field but as simple text.

Java method

Up to now, the list of transfers is empty. The transfers have been stored into the database but have not been read from there. Now the transfers will be loaded from the database into the data model.

- Extend the class *Banking* with the following methods in order to read bookings from the database.

```
...
import de.zeos.zen.api.fom.*;
import java.sql.ResultSet;
...

 public static List readBookings() {
  List buchungen = null;
  Connection con = null;
  PreparedStatement prepStmnt = null;
  ResultSet rst = null;
  try {
   con = new InitialServiceConnector().getConnectionService().getConnection(dbPool);
   prepStmnt = con.prepareStatement("select text, amount from entries");
   rst = prepStmnt.executeQuery();
   buchungen = createBookings(rst, "text", "amount", "entry", "entries");
  }
  catch (Exception e) {
   throw new OperationRuntimeException("Select on 'entries' failed", e);
  }
  finally {
   try {
    if (null != rst)
     rst.close();
    if (null != prepStmnt)
     prepStmnt.close();
    if (null != con)
     con.close();
   }
   catch (Exception e) {
    //jdbc error
   }
  }
  return buchungen;
 }

 private static List createBookings(ResultSet rst, String textName, String amountName,
                           String entryName, String entriesName) throws Exception {
  ElementBuilder builder = ElementBuilder.getInstance();
  List entries = builder.newList(entriesName);
  while (rst.next()) {

   Atom text = builder.newAtom(textName);
   text.setData(rst.getString(textName));

   Atom amount = builder.newAtom(amountName);
   amount.setData(new Double(rst.getDouble(amountName)));

   // create composition of atoms
   Composition entry = builder.newComposition(entryName);
   entry.setElement(text);
   entry.setElement(amount);

   // add to List
   entries.addElement(entry);
  }
  return entries;
 }
```

Save the class. These methods read the bookings from the database table *entries,* populate a composition for each entry and add them to the list *entries*. The method *readBookings* will now be created as workflow operation in the *Business Logic View* and will be connected to the data model.

- Create a new operation in the *Business Logic View* on the tab *Workflow* using the context menu below.
- Enter `read bookings from database` as description and select the method `readBookings` from the class `Banking`.

This method has no input argument but the list of transfers as return value.

- Unfold the model tree until you are able to select the `entries` list.
- Click on the lower right arrow button. This assigns the return value to the element.

As final step you must define when this operation should be executed.

- Activate the *Action View*, select the action `show` and add the operation `read bookings from database` on the tab *Operations*.

This causes the operation to be executed each time the action *show* is triggered. All previously saved transfers will be loaded into the data model and displayed on the state node *transfers*.

- Save the model and test the new functionality (see Figure 44). In the installed database, there are already two sample transfers.

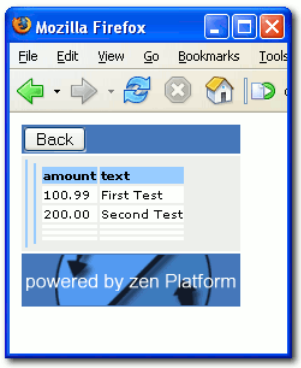

Figure 44
Browser: The state node „transfers“

8 Summary

We hope that this tutorial showed you by which simple steps a application can be developed, tested and extended in very short time using the *zen Platform*.

It showed just a small fraction of all the functionality provided by the *zen Platform*. Things like localization, transparent scaling, the service API or variable In-/Output formats have not been mentioned in this very simple show case.

For further information please refer to the *zen Platform Developers Guide*. It explains all functionality needed to develop a complete application.

If you have encountered problems during this tutorial or have further questions regarding the *zen Platform* please contact us at support@zeos.de.

Appendix

A Additional Reading

I. zen Platform

The complete documentation on the *zen Platform* can be found in the *zen Platform Developers Guide* which can be downloaded from (currently German only)

http://www.zeos.de/support/download/zen-Developers-Guide.pdf

II. Eclipse

There are several good sources and tutorial about Eclipse on the web.

Eclipse Homepage

The Eclipse-Homepage provides information about Eclipse and its sub projects.

http://www.eclipse.org

Eclipse Project FAQ

This FAQ answers several questions about Eclipse.

http://www.eclipse.org/eclipse/faq/eclipse-faq.html

Eclipse Documentation

Shortcut to the official Eclipse-Documentation.

http://www.eclipse.org/documentation/main.html

Eclipse Community

A link list providing a lot of information, resources and plugins related to Eclipse.

http://www.eclipse.org/community/main.html#EclipseInformation

B FAQ – Frequently Asked Questions

Please refer to the current release notes as well. You can find them in the directory *plugins/de.zeos.zenit_2.x.y* of your Eclipse installation.

Q: On startup of the *zen Engine* there is a error message: `...java.net.BindException: Address already in use:8989 ... Deployment shutdown`.

A: The *zen Engine* is already running on port 8989. Once running, there is no need to start it again. If you want to start a new instance of the *zen Engine*, simply stop the currently running instance before. See Chapter 4.6.
In very rare cases another software might use the port 8989. You can change the port used by the *zen Engine* by changing the value of the property `de.zeos.init.catalina.port` in the file *zen.properties* located in the directory *conf/* of your project.

Q: Everytime I'm trying to enter a value in a certain Eclipse view of the *zen Platform*, my entry is dismissed and reset it to the previous value or it disappears.

A: Probably you tried to enter an illegal value for this field. Watch out for any error messages on the lower left edge of the Eclipse window. Error messages will be displayed during input in this status line.

Q: If I'm testing pages with the browser several times, the navigation does not work as expected and it displays the same page over an over again.

A: You propably used the *"Back"* button of your browser. To provide 100% consistent data, the *zen Engine* only permits transitions that have been explicitly defined. Define a transition if you want provide this kind of navigation or close the browser and start it again.

Q: The browser shows an unexpected result. Labels are missing or the workflow differs from what you have defined.

A: Have you saved all your changes, including changes to Java files? Watch out for any misspelling or upper-/lowercase issues, especially error states like *builtin:error* and *builtin:timeout,* the label names *headline*, *label* and *error* and the return values of decisions have to be spelled exactly in the given way.

Q: The browser displays an error like *`The connection was refused when attempting to contact localhost:8989`* .

A: Probably the *zen Engine* is not running.
Startup is explained in Chapter 4.6

Q: The Browser displayes an error state. The log output contains an error like:: `Error producing xml output!; nested exception is: No value found for resource 'label' with locale`

A: Some resources have not been translated to the locale the application is running. Change the zen Developer to the desired locale and activate the Resource View. All entries displayed with `<not localized>` should be translated.