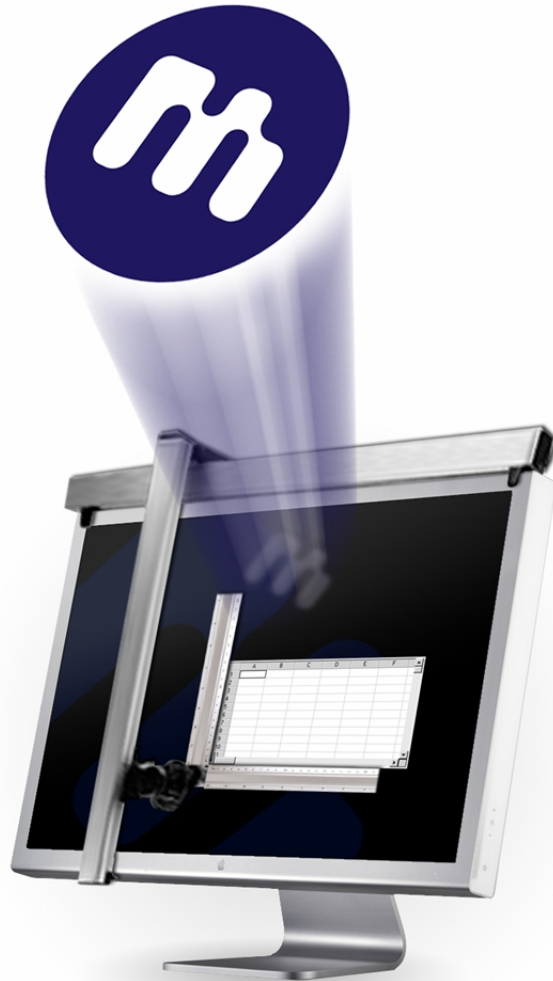# IT Mill Toolkit

Building a Simple Address Book

IT Mill Toolkit

IT Mill Toolkit Application Tutorial: Building a Simple Address Book

IT Mill Toolkit 5.3.

Published: 2009-03-05

Abstract

The purpose of this tutorial is to give an overall idea of what development with IT Mill Toolkit is like. We'll explore some basic patterns and how to build desktop style layouts with best practices.

IT Mill Toolkit

Table of Contents

List of Figures

Preface

1. About This Tutorial

This tutorial is intended for software developers learning to use IT Mill Toolkit for developing web applications.

1.1. Prerequisites

This book assumes that you have some experience with programming in Java. If not, Java is easy to learn if you have experience with other object oriented languages, such as C++. You may have used some desktop-oriented user interface toolkits for Java, such as AWT, Swing, or SWT. Or for C++, a toolkit such as Qt. Such knowledge is useful for understanding the scope of the Toolkit, but not necessary. Regarding the web, it is good if you know the basics of HTML and CSS, so that you can develop basic presentation themes for the application. Knowledge of Google Web Toolkit (GWT), JavaScript, and AJAX is needed only if you develop special custom UI components.

1.2. Online Resources

Developer's Site

The IT Mill Toolkit Developer's Site [http://dev.itmill.com/] provides various online resources, such as a development wiki, ticket (bugs and other issues) management system, source repository browsing, timeline, development milestones, and so on.

- Checkout IT Mill Toolkit source code from the Subversion repository

- Read technical articles and get more examples

- Report bugs

- Make requests for enhancements

- Follow the development of the Toolkit

- Collaborate!

The wiki provides instructions for developers, especially for those who wish to checkout and compile IT Mill Toolkit itself from the source repository. The technical wiki articles deal with integration of IT Mill Toolkit applications with various systems, such as JSP, Maven, Spring, Hibernate, and portals. The wiki also provides answers to Frequently Asked Questions.

Online Documentation

You can read this book online at http://www.itmill.com/documentation/. You can find technical articles and answers to Frequently Asked Questions also from the Developer's Site [http://dev.itmill.com/].

1.3. Support

Support Forum

An open support forum for developers is available at http://forum.itmill.com/. Please use the forum to discuss any problems you might encounter, wishes about features, and so on.

- Share your ideas and code

- Ask and you get answers

- Search answers from archived discussions

Bug Report Form

If you have found an issue with IT Mill Toolkit, demo applications or documentation, please report it to us by filing a ticket in the IT Mill Toolkit developer's site at http://dev.itmill.com/. You may want to check the existing tickets before filing a new ticket. You can make a ticket to make a request for a new feature as well, or to suggest modifications to an existing feature.

Commercial Support

IT Mill offers full commercial support and training services for the IT Mill Toolkit products. Please contact our sales at http://www.itmill.com/itmill_contact_sales.htm for details.

Chapter 1. Introduction

The purpose of this tutorial is to give an overall idea of what development with IT Mill Toolkit is like. We'll explore some basic patterns and how to build desktop style layouts with best practices.

We will extend components to make further development snappier, using forms and tables efficiently together with the data binding features available in IT Mill Toolkit.

Note that the application we build during this tutorial is not exactly what we call a "perfectly engineered" IT Mill Toolkit application. We have left some features out to keep things simple enough. These include:

- Best possible usability (Usability design)

- Localization / strings are hard coded

- Code organization

- Multi-user support (login)

- Persistency, no ORM or basic database usage

What we will be creating in this tutorial is an address book application with a main view like the following:

Figure 1.1. What We Aim At
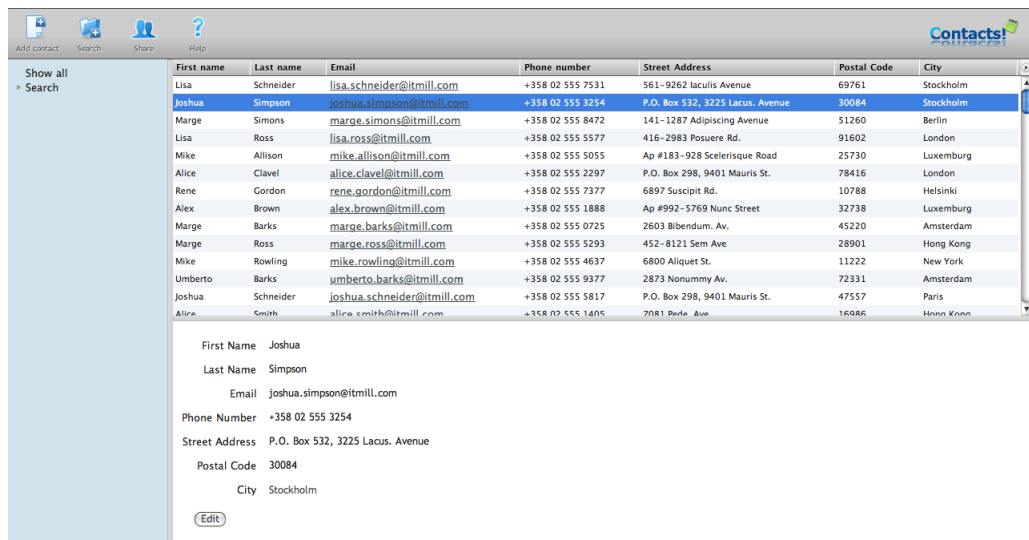
The program has a quite common layout. At the top there is a toolbar containing a few shortcut buttons and a logo. On the left there is a tree used for navigation and also for showing some data (saved searches). The rest of the area (on the right hand side) contains the current view in the application. As an address book is a rather simple example, we'll only build two views for our application. The one in the screenshot above (the **Show all** view) contains a table and a detailed viewer/editor of the selected row. The other view is a simple search view, shown in Figure 1.2, "Search View" below:

Figure 1.2. Search View
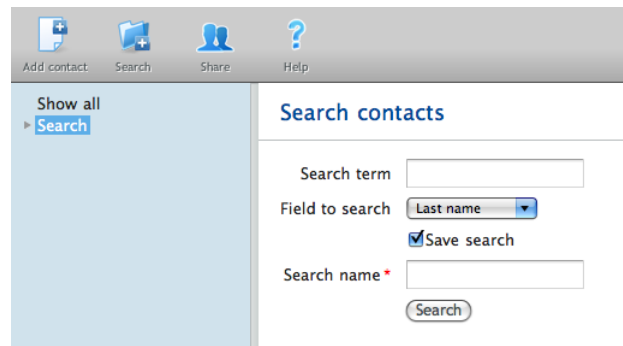
Some features implemented in the application:

- A toolbar with shortcut buttons

- A multi-feature navigation tree

  - **Show All** opens the main view

  - **Search** opens the search view

  - Quick links for saved searches below "**Search**"

- A main view containing a list of contacts

- A details view + editor for updating an existing contact

- Adding of new contacts

- Simple search features, with the ability to save common searches

This tutorial consists of six steps (chapters 2-7). We have prepared an Eclipse project package of the project state after each of the steps. The packages are Zip files. Download and uncompress the package you want to import, and import it to Eclipse. The packages are available for download at:

- http://www.itmill.com/download/doc/tutorial/itmill-toolkit-5-tutorial/

Chapter 2. Project setup

Before we get started we need to setup the following things:

- Eclipse IDE or any other IDE of your choise. Eclipse an be downloaded from the eclipse site. Choose Eclipse IDE for Java EE Developers.

- A servlet container like for instance Tomcat.

- The IT Mill Toolkit package - this tutorial is written with the 5.3 series so choose the latest Release Candidate until 5.3 is finally released.

We will assume below that you are using Eclipse and Tomcat but anything similar is fine.

Start up your Eclipse and choose **File** → **New** → **Project**.

Figure 2.1. New Project Window in Eclipse
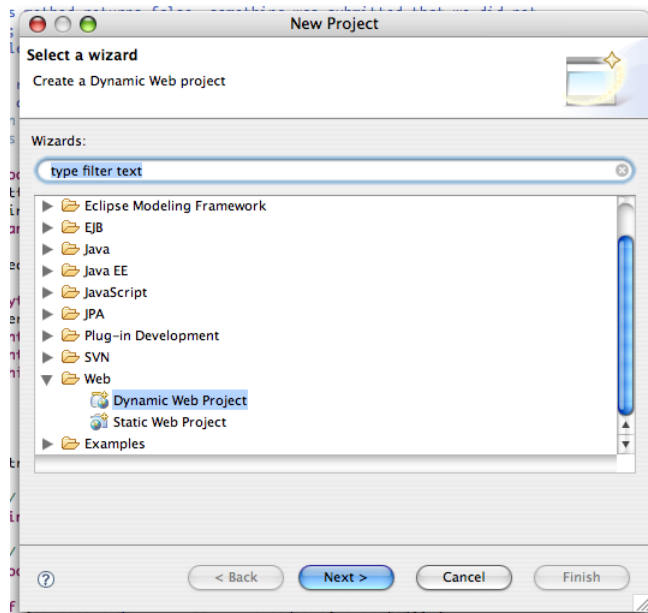
We will be creating a Dynamic Web Project which is designed to build web applications deployed as a .war file to a servlet container.

Figure 2.2. New Dynamic Web Project Window in Eclipse
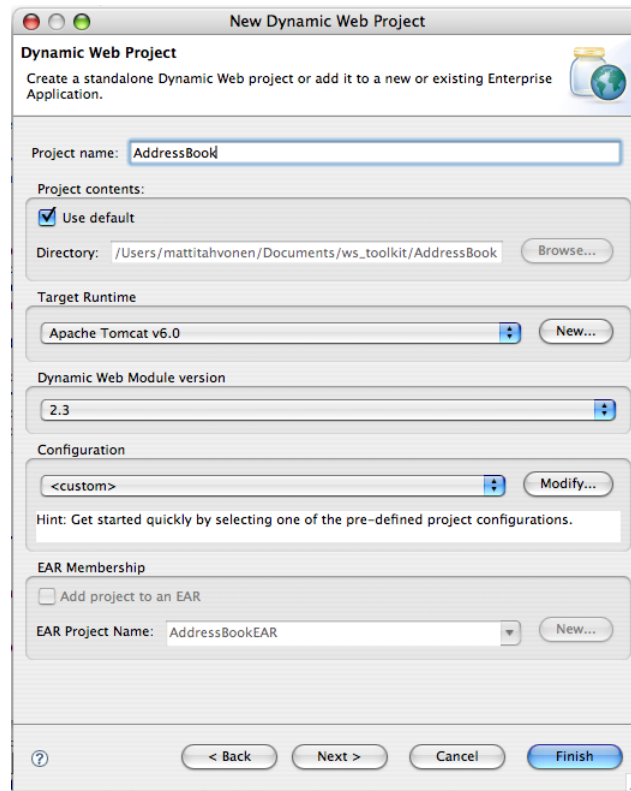
Add the following settings to the project:

- **Project name**

- **Target runtime**

  - **Choose new**

  - Select Apache Tomcat v6.0

  - check **Create a new local server**

  - Choose your Tomcat directory in the next step

- Click **Finish** and switch to the suggested Java EE mode

The project creator will automatically create a proper hierarchy for our project.

Finally open the Java EE perspective (Eclipse asks if you want to do this).

Next we need to add the Toolkit library to our project. Drop the earlier downloaded `itmill-toolkit-5.3.0*.jar` into the `WebContent/WEB-INF/lib` directory.

That is it, now we are ready to create the main class for our application. Right click on the project in the Project Explorer and choose **New → Class**.

Figure 2.3. New Java Class Window in Eclipse
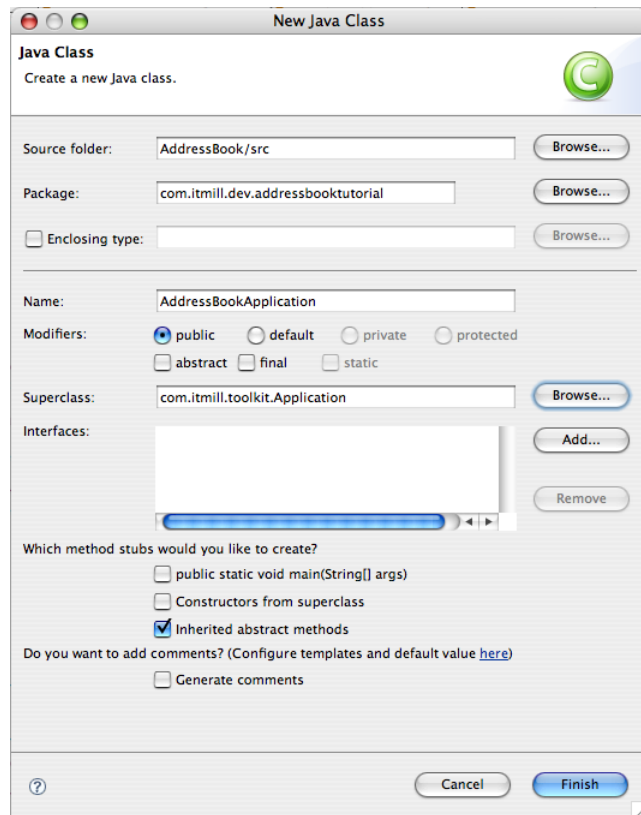
Type in:

- a proper package name (com.itmill.dev.addressbooktutorial)

- a name for the class (**AddressBookApplication**)

- We'll be extending **com.itmill.toolkit.Application** so choose that as the super class

Next, click **Finish** and Eclipse will create the new class for you.

Let's first ensure that we get basic things like deployment and debugging working. Change your newly created application class to contain the following "Hello world" code for testing purposes:

```
package com.itmill.dev.addressbooktutorial;

import com.itmill.toolkit.Application;
import com.itmill.toolkit.ui.Label;
import com.itmill.toolkit.ui.Window;

public class AddressBookApplication extends Application {

    @Override
    public void init() {
        setMainWindow(new Window("Address Book"));
        getMainWindow().addComponent(new Label("Hello World"));
    }
}
```

Hint 1

If the formatting is weird after copy/pasting code from this tutorial, press **Ctrl**-**Alt**-**F** to let Eclipse format the code.

Hint 2

If you copy/paste some code and Eclipse doesn't recognize some of the classes, press **Ctrl**-**Shift**-**O** to automatically organize the imports.

Double click in the left margin next to the `init()` method to add a break point.

To run our application in Tomcat we need to do some changes to our web.xml file. This is the only XML configuration file you need to touch! We need to define the Toolkit application servlet which responds to browser requests. The Eclipse project creator has created a basic `web.xml` for us in the `WebContent/WEB-INF/` directory. Open it in the text editor and modify it so it looks like the following. Make sure there is no white space before `<?xml` or Eclipse will complain about invalid XML.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
                         "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app id="WebApp_ID">
  <display-name>AddressBook</display-name>

  <!-- ApplicationServlet from Toolkit jar will respond to requests. -->
  <servlet>
    <servlet-name>AddressBook</servlet-name>
    <servlet-class>
        com.itmill.toolkit.terminal.gwt.server.ApplicationServlet
    </servlet-class>
    <init-param>
      <!-- Define our application class for servlet. -->
      <param-name>application</param-name>
     <param-value>com.itmill.dev.addressbooktutorial.AddressBookApplication</param-value>

    </init-param>
  </servlet>

  <!-- Make the servlet respond to all requests. -->
  <servlet-mapping>
    <servlet-name>AddressBook</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>
</web-app>
```

See the *Reference Manual* for an explanation as to why the web.xml file looks like this. Now we are ready to deploy our project to Tomcat. Right click on the AddressBook project in Project Explorer and choose **Debug as** → **Debug on Server**. All settings in the popup ought to be OK so just click **Finish**. Eclipse will now start Tomcat and deploy the project.

Before the application shows up the debugger should hit the `init()` method and Eclipse asks you if you want to move automatically to the debug perspective. Do that and you can debug your application code just as you would debug any application code in Eclipse. When ready, remove the break point by double clicking next to `init()` again and then click **Play** to resume execution.

When starting the application, Eclipse automatically opened an embedded web browser in a new tab for you (it should say "Address Book"). Open the tab and ensure you see "Hello World" on your screen. Next you can copy/paste the URL to your favorite web browser to verify it works from there as well.

If everything worked, we are ready to start building the main layout. If there was a problem, go back and recheck the steps or download the Eclipse project for the next step and start from there. If you ran into problems then also check the *IT Mill Toolkit Reference Manual* for a Quick Start guide on how to install Tomcat.

Note

Note: After this step we have done ALL the needed configurations to start developing IT Mill Toolkit based applications. No more libraries needed, no more XML configurations needed whatsoever - no matter how much you extend your application in the future.

You can download the Eclipse project package for this step from:

- http://www.itmill.com/download/doc/tutorial/itmill-toolkit-5-tutorial/ab-ch2-setup.zip

Chapter 3. Application Skeleton

We are now ready to start building the actual Application, starting from the layout. Our component tree in the final application will look as shown in Figure 1.1, "What We Aim At" earlier.

- **Window**

  - **VerticalLayout** (every window has a layout)

    - **HorizontalLayout** (our toolbar)

      - **Button** (**Add contact**)

      - **Button** (**Search**)

      - **Button** (**Share**)

      - **Button** (**Help**)

    - **SplitPanel** (Horizontal)

      - **Tree** (navigation tree on left side)

      - Main view area (on the right side, contents depends on the application state)

We will build the main layout in the main application class but separate larger UI parts to their own, distinct classes. This is a best practice in general as well, it is always good to componentize similar things so that they (1) can easily be reused and (2) so that they are easier to understand and maintain.

3.1. The Application Class

The `init()` method in our application is called every time a new application needs to be initialized i.e. when a new user navigates to our application. To keep the `init()` method simple we will not build our main layout there but in a separate method (`buildMainLayout()`) which we call from `init()`. So let us start by creating the method called `buildMainLayout()` and adding a call to that method from the `init()` method. The code should look like this:

```
@Override
public void init() {
    buildMainLayout();
}

private void buildMainLayout() {
    setMainWindow(new Window("Address Book Demo application"));
}
```

Next we'll add some components which should be on the screen all the time. We'll store these in fields as we need to reference them later on.

```
private Button newContact = new Button("Add contact");
private Button search = new Button("Search");
private Button share = new Button("Share");
private Button help = new Button("Help");
private SplitPanel horizontalSplit = new SplitPanel(
        SplitPanel.ORIENTATION_HORIZONTAL);
```

3.2. Building the Main Layout

Next we'll create a **VerticalLayout**, the layout for our main window and make it consume all of the available space in the window. Then we create a new **HorizontalLayout** for the toolbar and populate it with our buttons. Finally we add the horizontal split panel to the window's main layout. We have separated the creation of the toolbar to a method of its own, `createToolbar()`, to keep the code cleaner. We end up with the following code:

```
private void buildMainLayout() {
    setMainWindow(new Window("Address Book Demo application"));
    VerticalLayout layout = new VerticalLayout();
    layout.setSizeFull();

    layout.addComponent(createToolbar());
    layout.addComponent(horizontalSplit);

    /* Allocate all available extra space to the horizontal split panel */

    layout.setExpandRatio(horizontalSplit, 1);
    /* Set the initial split position so we can have a 200 pixel menu to the left */


    horizontalSplit.setSplitPosition(200, SplitPanel.UNITS_PIXELS);

    getMainWindow().setLayout(layout);

}

public HorizontalLayout createToolbar() {

    HorizontalLayout lo = new HorizontalLayout();
     lo.addComponent(newContact);
     lo.addComponent(search);
     lo.addComponent(share);
     lo.addComponent(help);

    return lo;

}
```

Hint!

If Tomcat is running when you alter the class (copy/paste code) you might get a "Hot Code Replace Failed" popup. Just click **Terminate** to stop tomcat and then start the application again when you want to have a look at the application.

Try it out and you'll see the buttons at the top and the split panel at the bottom. The `setExpandRatio()` call makes the split panel use all available space that the toolbar doesn't want.

Figure 3.1. Initial Skeleton
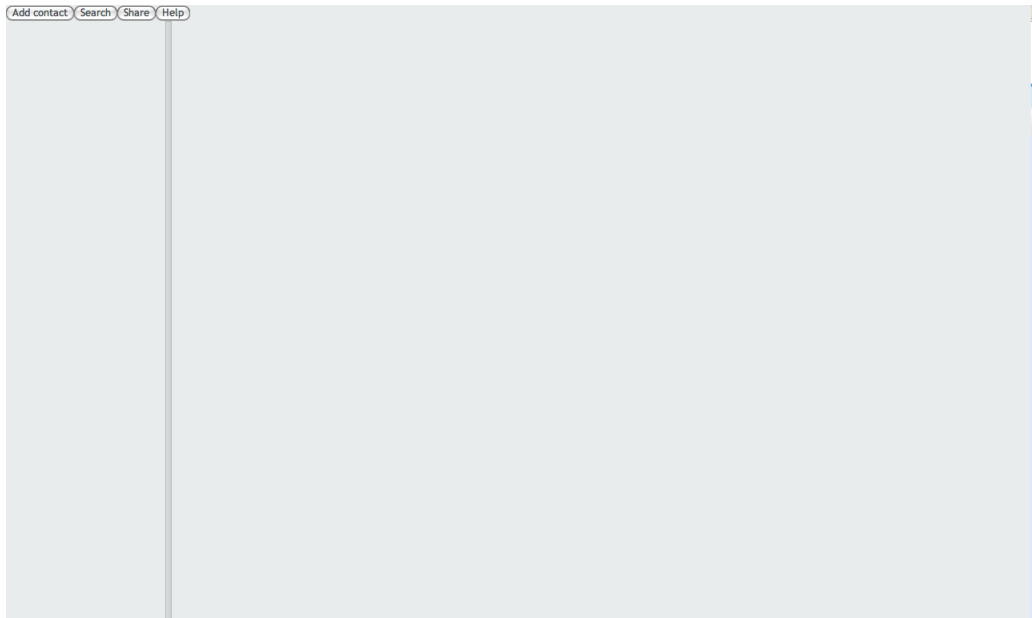
3.3. Populating Application With Components

Now we have a very basic application skeleton ready. The rest of the UI parts we need will be created by extending existing components. We'll extend basic components and tune them for use in our application. Let's start with a navigation **Tree**.

Create a **NavigationTree** class extending the Tree component in a new package called "ui". Eclipse will do most of this for you almost automatically. Add two lines of code to **Tree** in the constructor. You'll end up with the following class:

```
package com.itmill.dev.addressbooktutorial.ui;

import com.itmill.toolkit.ui.Tree;

public class NavigationTree extends Tree {
    public static final Object SHOW_ALL = "Show all";
    public static final Object SEARCH = "Search";

    public NavigationTree() {
            addItem(SHOW_ALL);
            addItem(SEARCH);
    }
}
```

Hint!

Code for a complete class like the above can be pasted directly into the AddressBook project in the Eclipse Package Explorer provided you are using the Java perspective. Eclipse automatically places the file in the correct place.

Now lets add the navigation tree to our application class. We'll need to reference it later so we store it in a field:

```
private NavigationTree tree = new NavigationTree();
```

Then we set it is as the first component for the horizontal split panel in `buildMainLayout()`.

```
horizontalSplit.setFirstComponent(tree);
```

The next step is to create our views. As in any desktop style UI framework in Toolkit there is no such a thing as a page (contrary to HTML/JSP which is page based). Instead we work with views just as in desktop applications. The application's main modes is what we usually call views. In our example application we only have two views. A search view and a contacts list view. If you are more familiar with page centric web application technologies like JSP or PHP, you might consider views as a kind of page. Or try not to think of creating something for the web, but just create a desktop application that then happens to run in a browser. Views may be constructed in a many different ways, but we are using the simplest approach here: our views are simply extended Toolkit components.

In our application the main view will always use the right side of the horizontal split panel. Let's start by making a generic setter for the main view in the **AddressBookApplication**:

```
private void setMainComponent(Component c) {
    horizontalSplit.setSecondComponent(c);
}
```

We want this to be a separate method so we can use it later on to change the main component (the view).

Now we create a **ListView**, which is our main view that is shown when the application is started. We create it in the same manner as the **NavigationTree** but extend **SplitPanel** instead of **Tree**. This split panel should be vertical, which is the default for split panels. We only need an empty class with a constructor so far.

```
package com.itmill.dev.addressbooktutorial.ui;

import com.itmill.toolkit.ui.SplitPanel;

public class ListView extends SplitPanel {
    public ListView() {
    }
}
```

To minimize start-up time and memory usage we will use a lazy initialization pattern for creating our views. In an application this small it doesn't actually matter, but generally it is a very good habit to lazily instantiate you GUI objects. So create a field for the ListView in the main application class and create a lazy getter for it like this:

```
private ListView listView = null;

private ListView getListView() {
    if (listView == null) {
        listView = new ListView();
    }
    return listView;
}
```

The method creates the view when it is needed for the first time, only creating the views actually used. Now test this by changing the code in the `init()` method to:

```
setMainComponent(getListView());
```

We'll be using this pattern for the most of our UI code.

Now let's create the components we want to put in the **ListView**. The first component (contacts list) will be extended from Table so we create a sub class of **Table** called **PersonList**. In its constructor we'll add some dummy data to it and also set the size to make it consume all available space given to it by the split panel. We end up with the following stub class:

```
package com.itmill.dev.addressbooktutorial.ui;

import com.itmill.toolkit.ui.Table;

public class PersonList extends Table {
    public PersonList() {
        // create some dummy data
        addContainerProperty("First Name", String.class, "Mark");
        addContainerProperty("Last Name", String.class, "Smith");
        addItem();
        addItem();
        setSizeFull();
    }
}
```

Next we create the viewer/editor for our contacts (**PersonForm**) in the same way by extending **Form**:

```
package com.itmill.dev.addressbooktutorial.ui;

import com.itmill.toolkit.ui.Button;
import com.itmill.toolkit.ui.Form;
import com.itmill.toolkit.ui.HorizontalLayout;
import com.itmill.toolkit.ui.TextField;

public class PersonForm extends Form {

    private Button save = new Button("Save");
    private Button cancel = new Button("Cancel");

    public PersonForm() {
        addField("First Name", new TextField("First Name"));
        addField("Last Name", new TextField("Last Name"));
        HorizontalLayout footer = new HorizontalLayout();
        footer.setSpacing(true);
        footer.addComponent(save);
        footer.addComponent(cancel);
        setFooter(footer);
    }
}
```

Modify the **ListView** constructor so we can pass the components to it and add them to the split panel. Modify the **ListView** getter to instantiate those new components too and save a reference to them. You'll end up having the following code in your application class:

```
private PersonList personList = null;
private PersonForm personForm = null;

private ListView getListView() {
    if (listView == null) {
        personList = new PersonList();
        personForm = new PersonForm();
        listView = new ListView(personList, personForm);
    }
    return listView;
}
```

And the following in the **ListView** constructor:

```
package com.itmill.dev.addressbooktutorial.ui;

import com.itmill.toolkit.ui.SplitPanel;

public class ListView extends SplitPanel {
    public ListView(PersonList personList, PersonForm personForm) {
        setFirstComponent(personList);
        setSecondComponent(personForm);
```
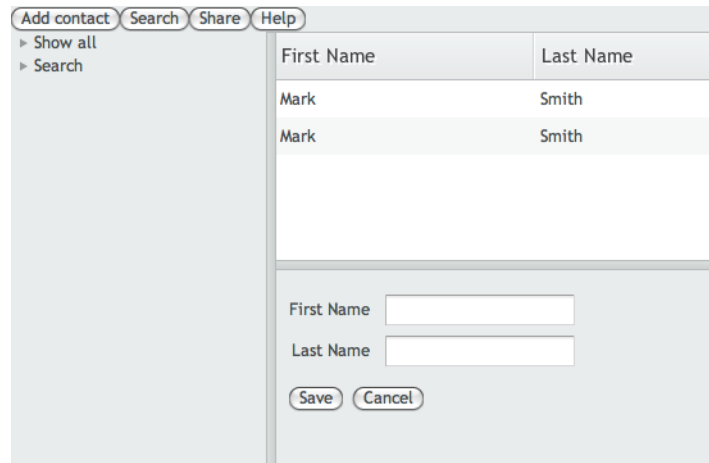
```
        setSplitPosition(40);
    }
}
```

The application should now look like this:

Figure 3.2. Skeleton with Views

3.4. Sub Windows

As the last step in this chapter we'll get to know an often used feature in IT Mill Toolkit: *sub windows*. Toolkit has two kinds of windows: top level windows and sub windows.

Traditionally windows are not that much used in web applications, but during the "Web 2.0" era the kind of windows inside browser windows have become quite common. Sub windows are added to top level windows (~web browsers windows). With moderate usage of sub windows one can make the application feel more like a desktop application and really improve usability.

Sub windows are shown in the application by adding them to a top level window using the addWindow(Window window) method. Most commonly like the following:

```
getMainWindow().addWindow(mySubWindow);
```

We'll make two sub windows for our application. One is a help window which will be floating over the main UI. The user can drag it to some place where it doesn't disturb application usage and resize it so it is not in the way. The other sub window is a (non-functional) **Sharing options** screen. It will demonstrate how to create a modal dialog that blocks the rest of the UI until it is closed. Both of those windows extend **Window** and are created much in the same way as **ListView** and **SearchView** above. Implementations are below:

```
package com.itmill.dev.addressbooktutorial.ui;

import com.itmill.toolkit.ui.Label;
import com.itmill.toolkit.ui.Window;

public class HelpWindow extends Window {
    private static final String HELP_HTML_SNIPPET = "This is "
            + "an application built during <strong><a href=\""
            + "http://dev.itmill.com/\">Toolkit</a></strong> "
            + "tutorial. Hopefully it don't need any real help.";

    public HelpWindow() {
```

```
                setCaption("Address Book help");
                addComponent(new Label(HELP_HTML_SNIPPET, Label.CONTENT_XHTML));
        }

}

package com.itmill.dev.addressbooktutorial.ui;

import com.itmill.toolkit.ui.Button;
import com.itmill.toolkit.ui.CheckBox;
import com.itmill.toolkit.ui.Label;
import com.itmill.toolkit.ui.Window;

public class SharingOptions extends Window {
    public SharingOptions() {
        /*
         * Make the window modal, which will disable all other components while
         * it is visible
         */
        setModal(true);

       /* Make the sub window 50% the size of the browser window */
        setWidth("50%");
        /*
         * Center the window both horizontally and vertically in the browser
         * window
         */
        center();

        setCaption("Sharing options");
        addComponent(new Label(
                "With these setting you can modify contact sharing "
                        + "options. (non-functional, example of modal dialog)"));
        addComponent(new CheckBox("Gmail"));
        addComponent(new CheckBox(".Mac"));
        Button close = new Button("OK");
        addComponent(close);
    }
 }
```

Now create lazy getter methods for both of those floating "views". We cannot try them out in the same way as the other views as we cannot add a window as a component. Instead we can test them using the `Window.addWindow()` method like this:

```
getMainWindow().addWindow(getHelpWindow());
```

In this step we have covered how to construct the main layout for our application. We created components for navigating the program, both in the form of buttons and a navigation tree. We have also created the table which will list all our contacts and created a form for inputting data. Finally we have introduced sub windows into our application and tested manually that everything works as expected. Next we need to add some logic to our application to make things happen when buttons are pushed etc. But before that we will see how to bind some data to our table.

You can download the Eclipse project package for the end of this step from:

- http://www.itmill.com/download/doc/tutorial/itmill-toolkit-5-tutorial/ab-ch3-layout.zip

Chapter 4. Data Binding Basics

We have now created the basic interface for our address book application and a table (**PersonList**) where all the contacts should be displayed. The next step is to bind data to the **PersonList** to make it show something more than the two "Mark Smith" test rows we added in the previous step.

4.1. Creating an Object

We want to base our table data on a simple **Person** java object containing fields such as `firstName`, `lastName`, `email`, etc. We do not want to populate the table by hand - instead we want to use Person instances for populating the table. The actual **Person** instances can come from anywhere, in a real case they would probably be fetched from a database but this is out of scope for this tutorial. We will instead generate random **Person**-objects and use them for demonstrating the data binding features.

The **Person** class we are using looks like:

```
package com.itmill.dev.addressbooktutorial.data;

public class Person {
        private String firstName = "";
        private String lastName = "";
        private String email = "";
        private String phoneNumber = "";
        private String streetAddress = "";
        private Integer postalCode = null;
        private String city = "";

// + setters and getters for all fields
}
```

Hint!

The getters and setters can be generated automatically by Eclipse: press Alt-Shift-S and choose **Generate Getters and Setters**.

4.2. Basics of the Data Model

When populating a table there are a few terms that must be understood. Each component that includes data (e.g. a **Table**) has a **Container**. The container includes all the **Item**s in the component. An **Item** corresponds to a row in the table. An **Item** has *properties* (**Property**) which in a table correspond to the columns in a row.

In simple applications we can use the `addItem()`-methods in the **Table** to populate it with data. The `addItem()`-methods add items (rows), which then in turn can be populated by adding properties and setting values for them - much like you would populate a traditional HTML table by creating <TR> and <TD> tags.

The data model in IT Mill toolkit also contains much more powerful features than this. Each table is connected to a data source which actually contains all the data displayed in the table. A data source can actually supply data for any type of component like a **Table**, **Select**, **Tree**, etc. The data source does not necessarily contain all the items for the table but can dynamically supply the table with data from another source when the **Table** needs it (for instance from a database). The data source is what we in IT Mill toolkit call a **Container**.

In this case we want to use a container which is based on a list of our **Person** objects, supplying the table with information directly from the objects when needed. The **BeanItemContainer** we will be using has not yet made its way into the official distribution but can be downloaded from http://dev.itmill.com/export/6618/incubator/beanitemcontainer/beanitemcontainer.jar. The downloaded jar-file should be placed in `WebContent/WEB-INF/lib` so it will be on the java classpath.

The nice thing with this container is that we can simply

- Instantiate the container

- Add a number of **Person** objects to it

- Assign the container to a table as a data source

- VOILA, the information is displayed in the table

There are also other **Container**s already in the official distribution, but nothing as powerful as the **Bean-ItemContainer**.

4.3. Creating a Custom Container

For this tutorial we will extend the **BeanItemContainer** and create a **PersonContainer** which contains randomly generated data:

```
package com.itmill.dev.addressbooktutorial.data;
import com.itmill.toolkit.data.util.BeanItemContainer;

public class PersonContainer extends BeanItemContainer<Person> {

    public PersonContainer() throws InstantiationException,
    IllegalAccessException {
        super(Person.class);
    }
}
```

In addition to this the **PersonContainer** contains a static method (`createWithTestData()`) for creating a container with test data. This consists of a loop where **Person** objects with random names and other data is generated and then added to the container using the `addItem(Person)`-method. The generation will not be described further in this tutorial and the `PersonContainer.java` can be downloaded from http://dev.itmill.com/export/6618/incubator/examples/addressbook/src/com/itmill/dev/addressbooktutorial/data/PersonContainer.java. For a real world application connected to a database we would simply replace this random data generation with a database query that fetches real objects. An example of how to do this with Hibernate can be found at http://dev.itmill.com/wiki/Articles/UsingHibernateWithToolkit.

4.4. Binding Table to a Container

To use this container in our application we add a new field to our **AddressBookApplication** together with a getter:

```
private PersonContainer dataSource = PersonContainer.createWithTestData();

public PersonContainer getDataSource() {
    return dataSource;
}
```
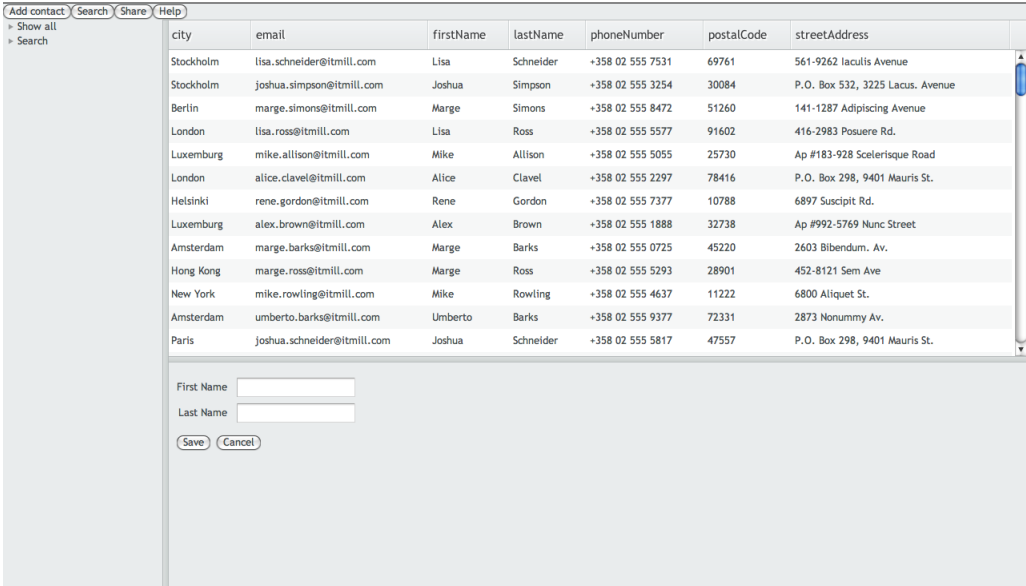
In addition to this we need to change the **PersonList** constructor so we pass the application instance to it (so it can acquire its data source). In addition we can remove the dummy test data creation from the previous step, leaving the **PersonList** class as:

```
public class PersonList extends Table {
      public PersonList(AddressBookApplication app) {
            setSizeFull();
            setContainerDataSource(app.getDataSource());
      }
}
```

The `setContainerDataSource()` sets the data source for the table to our own **PersonContainer**.

We now have a data source connected to the table and running the application will show that the **PersonList** contains 100 rows of data, based on 100 **Person**-objects which were randomly generated.

Figure 4.1. Connected Table

The order of the columns is still wrong, we want name to come first at least. Additionally the field names from the **Person** class aren't really human-friendly. We can handle this by adding two static arrays to the **PersonContainer** class (these will already be there if you downloaded the class).

```
public static final Object[] NATURAL_COL_ORDER = new Object[] {
          "firstName", "lastName", "email", "phoneNumber", "streetAddress",
          "postalCode", "city" };
   public static final String[] COL_HEADERS_ENGLISH = new String[] {
          "First name", "Last name", "Email", "Phone number",
          "Street Address", "Postal Code", "City" };
```

Additionally we add the following two rows to the **PersonList** class to sort the columns and make the headers nicer

```
setVisibleColumns(PersonContainer.NATURAL_COL_ORDER);
setColumnHeaders(PersonContainer.COL_HEADERS_ENGLISH);
```

The table will now look nicer:

Figure 4.2. Table Column Names

4.5. Summary

In this step we first created a **Person** class for the items that will be shown in the address book. Next we discussed **Container**s, **Item**s and **Property**-values - the central parts of databinding in Toolkit applications. We also introduced the **BeanItemContainer** and extended it for our own purposes. We created some dummy data and added it to the table by the methods discussed above. This is exactly the way you would bind e.g. data from a database or your system backend to the UI layer. Finally we also made table more readable and better looking. Next we will look at how the application starts to respond to user interactions.

You can download the Eclipse project package for the end of this step from:

- http://www.itmill.com/download/doc/tutorial/itmill-toolkit-5-tutorial/ab-ch4-data.zip

Chapter 5. Creating user interactions

At this point we have quite a lot of components on the screen and even a data source filled with test data attached to our table. It is time to start creating some real logic in our application. We'll start with a simple view navigation originating from user clicking on tool bar buttons, in the navigation tree or selecting an item in the table. We will then continue with some more advanced stuff in the form and the search view. But first of all we need to create a search view so we have something to navigate to. We will only create an empty view at this point to demonstrate navigation, the search components are added later.

```
package com.itmill.dev.addressbooktutorial.ui;

import com.itmill.dev.addressbooktutorial.AddressBookApplication;
 import com.itmill.toolkit.ui.Panel;

public class SearchView extends Panel {
    public SearchView(final AddressBookApplication app) {
        setCaption("Search contacts");
        setSizeFull();
    }
 }
```

We use the same kind of lazy initialization for this view as we did for the **ListView**.

```
    private SearchView searchView = null;

    private SearchView getSearchView() {
        if (searchView == null) {
            searchView = new SearchView(this);
        }
        return searchView;
    }
```

5.1. View navigation

In our application most of the event handling is done by the main application class. This is a working and a clean approach to hook control logic into applications of this size. There are several other methods to implement event handling like using inline listeners or separate listener classes. Patterns used in other GUI frameworks like Swing can easily be applied to an IT Mill Toolkit application so you can take advantage of your previous experiences in GUI programming.

The first step is to make navigation between the main view and the search work. To navigate to the search view we have a button in the toolbar. We'll start from there. Go to the createToolbar() method in the **AddressBookApplication** and add the following line close to the place you are adding the button to the layout.

```
search.addListener((Button.ClickListener) this);
```

Make **AddressBookApplication** implement **Button.ClickListener** and add the required buttonClickMethod() to the **AddressBookApplication** class. (Hint: you'll make this in about 2 seconds with Eclipse's quick fix feature, use Ctrl-1 (cmd-1 on mac) a few times).

Now that the application can listen to button click events, add listeners to all the other buttons created in createToolbar() as well. We'll later on use the same listener to handle their clicks too. Type in the following event handling code to the buttonClick() method:

```
final Button source = event.getButton();
if (source == search) {
    showSearchView();
}
```

Create the missing `showSearchView()` method (or let Eclipse create it for you). We want to keep the code clean so we keep the event handlers as small as possible and pass logic forward. This is essential especially if you are using listeners that will end up receiving events from multiple components.

The `showSearchView()` method contains the real logic which is extremely simple in this particular case: we'll just call the `setMainComponent()` method with `getSearchView()`. The `getSearchView()` method will do the lazy instantiation of the search view if necessary and `setMainComponent()` will replace the current view with the search view. When you have added the `showSearchView()` as below you are ready to test your program. Run it and click the search view to see that it works!

```
private void showSearchView() {
    setMainComponent(getSearchView());
}
```

Quite simple wasn't it? Enhance your skills by adding similar listener for **Share** and **Help**. You have learned to add sub windows in Section 3.4, "Sub Windows".

5.2. Person selection

Now lets concentrate on selecting a row in the table. First of all we need to set the table in selectable mode so the user can select rows. The value of a Table corresponds to the row that is currently selected which means that when a row is selected, a value change event is sent. A value change listener is thus a good place to react to user interaction. Add a value change listener to the table by adding the following code to the **PersonList** constructor:

```
        /*
         * Make table selectable, react immediatedly to user events, and pass events to
 the
         * controller (our main application)
         */
        setSelectable(true);
        setImmediate(true);
        addListener((Property.ValueChangeListener) app);
        /* We don't want to allow users to de-select a row */
        setNullSelectionAllowed(false);
```

Now go back to **AddressBookApplication** class. Make it implement **Property.ValueChangeListener** and add the required method `valueChange()`.

(Note that at this point you may have to add casts to your click listeners so that the `addListener()` method can decide which listener to add).

Now lets create or value change handler for the table. Our handler should simply retrieve the selected item from the table and pass it on to the **PersonForm** as a data source for it. If the datasource for the form already is this item we do not need to do anything. The `valueChange()` method should look like this:

```
    public void valueChange(ValueChangeEvent event) {
        Property property = event.getProperty();
        if (property == personList) {
            Item item = personList.getItem(personList.getValue());
            if (item != personForm.getItemDataSource()) {
                personForm.setItemDataSource(item);
            }
        }
    }
```

Now try it out! When clicking on a table row you ought to get the information for the selected **Person** displayed in the form below. The **Form** does not work correctly yet, but we'll get back to that later. The next step is to add listeners for our navigation tree.

5.3. Navigation tree

With our navigation tree we could do exactly the same kind of value change listener as we did for table. **Tree** is also a sub class of **Select**. For a tree used for navigation the value change event, however, is not the ideal event. Clicking on an already selected item in the tree will not fire a value change event but often it is necessary to react also to clicks on an already selected item. As this cannot be accomplished with a value change listener we use another kind of listener: **ItemClickListener**. **ItemClickListener**s are notified each time an item is clicked independent of the current selection.

Listening to item clicks is similar to listening to button clicks. We add a hook to the item container (**NavigationTree** in this case) and in the listener we can determine the item that was clicked on using the **ItemClickEvent** event. Steps to do:

- Let **AddressBookApplication** implement **ItemClickListener** and add the required method.

- Add **AddressBookApplication** as a constructor parameter to **NavigationTree** and add it as a **ItemClickListener** in the **NavigationTree** constructor. Although we are not interested in selections in a tree, make tree selectable also so the user gets some feedback when clicking on an item.

- Add logic to the the item click handler (`itemClick()`) to react to **Show all** and **Search**

After the changes the **NavigationTree** should look like this...

```
package com.itmill.dev.addressbooktutorial.ui;

import com.itmill.dev.addressbooktutorial.AddressBookApplication;
import com.itmill.toolkit.event.ItemClickEvent.ItemClickListener;
import com.itmill.toolkit.ui.Tree;

public class NavigationTree extends Tree {
    public static final Object SHOW_ALL = "Show all";
    public static final Object SEARCH = "Search";

    public NavigationTree(AddressBookApplication app) {
        addItem(SHOW_ALL);
        addItem(SEARCH);

        /*
         * We want items to be selectable but do not want the user to be able to
         * de-select an item.
         */
        setSelectable(true);
        setNullSelectionAllowed(false);

        // Make application handle item click events
        addListener((ItemClickListener) app);

    }
}
```

...and the `itemClick()` handler in **AddressBookApplication** should look like this:

```
    public void itemClick(ItemClickEvent event) {
        if(event.getSource() == tree) {
            Object itemId = event.getItemId();
            if (itemId != null) {
                if (itemId == NavigationTree.SHOW_ALL) {
                    showListView();
                } else if (itemId == NavigationTree.SEARCH) {
                    showSearchView();
                }
            }
```

```
        }
    }
```

The new `showListView()` simply calls `setMainComponent()` with `getListView()`. Try it out!

Got a **ClassCastException**? Remember that **AddressBookApplication** must implement **ItemClickListener**.

5.4. Improving the form

Next we'll concentrate on the **Form**. We want to edit the existing details. The value change listener in the table already puts the selected **Person** into the form. The values also get updated on the server. What we want to do is some usability improvements:

- Use the form as a detailed viewer (read only) in the first place and have an **Edit** button to enable editing.

- Put the form into a "buffered" mode so the form data gets submitted to data model only if **Save** is clicked. A **Cancel** button should return the form to the read-only state with the old values.

- Make the form fields appear in the same order as in the table.

The first step is to change the order of the fields so that it is the same as in the table. This makes it more natural to fill out the form. We can use the same method as with the table for ordering the fields:

Let's start from the **PersonForm** constructor and do the following modifications:

- remove dummy fields.

- add a main application reference via the constructor.

- call setWriteThrough(false) to enable buffering.

- make the form footer invisible by default (we don't want to show the footer when the form is missing its datasource i.e. not showing any Person).

- let the Form implement ClickListener and make it listen to the buttons in the footer.

- add one more button to the footer: "Edit". This will be visible when in read-only state.

After these modifications the **PersonForm** looks like the following. Note that you must also modify the main application class to pass itself in the **PersonForm** constructor call.

```
package com.itmill.dev.addressbooktutorial.ui;

import com.itmill.dev.addressbooktutorial.AddressBookApplication;
import com.itmill.toolkit.ui.Button;
import com.itmill.toolkit.ui.Form;
import com.itmill.toolkit.ui.HorizontalLayout;
import com.itmill.toolkit.ui.Button.ClickEvent;
import com.itmill.toolkit.ui.Button.ClickListener;

public class PersonForm extends Form implements ClickListener {

    private Button save = new Button("Save", (ClickListener) this);
    private Button cancel = new Button("Cancel", (ClickListener) this);
    private Button edit = new Button("Edit", (ClickListener) this);
    private AddressBookApplication app;

    public PersonForm(AddressBookApplication app) {
```

```
        this.app = app;

        // Enable buffering so that commit() must be called for the form
        // before input is written to the data. (Form input is not written
        // immediately through to the underlying object.)
        setWriteThrough(false);

        HorizontalLayout footer = new HorizontalLayout();
        footer.setSpacing(true);
        footer.addComponent(save);
        footer.addComponent(cancel);
        footer.addComponent(edit);
        footer.setVisible(false);

        setFooter(footer);
    }

    public void buttonClick(ClickEvent event) {

    }

}
```

Setting the form to buffering mode (setWriteThrough(false)) means that changes made by the user do not immediately update the datasource. Instead the datasource is updated when commit() is called for the form. We can then easily revert the data and get the old status back by calling discard(). This needs to be implemented in the button click handler:

```
    public void buttonClick(ClickEvent event) {
        Button source = event.getButton();

        if (source == save) {
            /* If the given input is not valid there is no point in continuing */
            if (!isValid()) {
                return;
            }
            commit();
            setReadOnly(true);
        } else if (source == cancel) {
            discard();
            setReadOnly(true);
        } else if (source == edit) {
            setReadOnly(false);
        }
    }
```

We can still not test our form as we have made the footer hidden so the form by default will be empty. Next we'll override two methods: setItemDataSource() and setReadOnly() to make the form work as we want it. The default implementations work just fine but we want to add some additional tasks to them.

The setItemDataSource() method is called when the datasource is changed, e.g. when the user selects a person in the table. At this point we want to do three things:

- set the form to a read-only state by calling setReadOnly(true). This is the initial state.

- ensure that the form footer is visible.

- make sure the fields are rendered in the order we want (consistent with the table)

In the setReadOnly() method we want to control which buttons are shown in the footer. **Save** and **Cancel** need to be visible when the form is in edit mode and the **Edit** button when the form is in read-only

mode. Calling `setItemDataSource()` will regenerate all fields inside the form so this is a good place to tell the form in which order we want the fields to be rendered.

Our overridden methods look like the following:

```
@Override
 public void setItemDataSource(Item newDataSource) {
     if (newDataSource != null) {
        List<Object> orderedProperties = Arrays
                .asList(PersonContainer.NATURAL_COL_ORDER);
        super.setItemDataSource(newDataSource, orderedProperties);

        setReadOnly(true);
        getFooter().setVisible(true);
     } else {
        super.setItemDataSource(null);
        getFooter().setVisible(false);
     }
 }

@Override
 public void setReadOnly(boolean readOnly) {
     super.setReadOnly(readOnly);
     save.setVisible(!readOnly);
     cancel.setVisible(!readOnly);
     edit.setVisible(readOnly);
 }
```

Try it out and ensure that changing mode (*read only* / *editable*) mode works correctly and that the values in the **PersonTable** are not updated until **Save** is clicked (when not using the buffering mode they are immediately updated). Remember to run **Organize Imports** in Eclipse after copy/paste editing.

5.5. Implement logic to add new contacts

Next we want to create new contacts. Most of this logic will go in to **PersonForm** but we must first create some logic in the main application class. Before passing control to the **PersonForm** for the actual addition we want to make sure the **ListView** is being shown and we also want to clear any possible selections from the table.

Add more logic to the click handler in **AddressBookApplication** and make a click on the **Add contact** button pass control to the `addNewContact()` method. Implement it as follows:

```
private void addNewContanct() {
     showListView();
     personForm.addContact();
 }
```

Next we create the `addContact()` method in **PersonForm**. Now the problem is that we do not have an existing **Person** object to edit but need to create a temporary one. We also need a flag that tells we are in "*newContactMode*" so we can add the **Person** to the datasource when the user clicks **Save** (we do not want to add it immediately as it would then show up in the **PersonTable** and additionally we would have to re-move it if the user clicks **Cancel**). We add two fields to accomplish this (could be combined but kept separate here for clarity):

```
private boolean newContactMode = false;
 private Person newPerson = null;
```

So we instantiate a new, temporary **Person** and put it into the form without affecting the datasource in the main application. This is done easily by wrapping the **Person** in a **BeanItem** and setting that as the data source for the form. We do not want a read only view when adding a contact so our `addContact()` method will look like this:

```
public void addContact() {
    // Create a temporary item for the form
    newPerson = new Person();
    setItemDataSource(new BeanItem(newPerson));
    newContactMode = true;
    setReadOnly(false);
}
```

Now try clicking on the **Add contact** button to ensure everything works this far. We are still lacking proper logic for handling **Save** and **Cancel** events for a new contact. When saving a new contact the temporary **Person** object, or actually the **BeanItem** wrapping it, must be added to our main data source, the **PersonContainer**. This will make the new contact show up in the table automatically. If the user clicks **Cancel** when adding a contact we must empty the form instead of setting it to read-only mode. With these changes our click listener in the form looks like this:

```
public void buttonClick(ClickEvent event) {
    Button source = event.getButton();

    if (source == save) {
        /* If the given input is not valid there is no point in continuing */
        if (!isValid()) {
            return;
        }

        commit();
        if (newContactMode) {
            /* We need to add the new person to the container */
            Item addedItem = app.getDataSource().addItem(newPerson);
            /*
             * We must update the form to use the Item from our datasource
             * as we are now in edit mode
             */
            setItemDataSource(addedItem);

            newContactMode = false;
        }
        setReadOnly(true);
    } else if (source == cancel) {
        if (newContactMode) {
            newContactMode = false;
            setItemDataSource(null);
        } else {
            discard();
        }
        setReadOnly(true);
    } else if (source == edit) {
        setReadOnly(false);
    }
}
```

Finally, "*newContactMode*" should also be cleared when selecting a new row in the table without clicking on **Save** or **Cancel**:
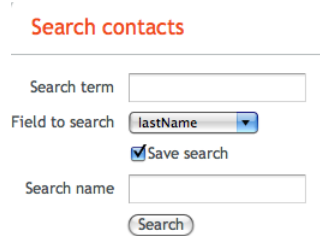
```
public void setItemDataSource(Item newDataSource) {
    newContactMode = false;
    ...
}
```

Try it out and check that both creation of new contacts and editing of old ones works correctly.

5.6. Implementing the search functionality

We have previously added logic for the search button so a search view is displayed when the button in the toolbar is clicked but the view is still empty. We want to create a simple view where you can enter a search term, select what field to match the term to and optionally save the search for later usage. The finished view should look like the following.

Figure 5.1. Search User Interface

We implement this using a **FormLayout** which will cause the captions to be rendered to the left of the fields instead of above (as the case is with a **VerticalLayout**). In addition to the layout we need a couple of **TextFields**, a **NativeSelect**, a **CheckBox** and a **Search** button. The following code is used for creating the layout.

```
package com.itmill.dev.addressbooktutorial.ui;

import com.itmill.dev.addressbooktutorial.AddressBookApplication;
import com.itmill.dev.addressbooktutorial.data.PersonContainer;
import com.itmill.toolkit.ui.Button;
import com.itmill.toolkit.ui.CheckBox;
import com.itmill.toolkit.ui.FormLayout;
import com.itmill.toolkit.ui.NativeSelect;
import com.itmill.toolkit.ui.Panel;
import com.itmill.toolkit.ui.TextField;
import com.itmill.toolkit.ui.Button.ClickEvent;
import com.itmill.toolkit.ui.Button.ClickListener;

public class SearchView extends Panel {

    private TextField tf;
    private NativeSelect fieldToSearch;
    private CheckBox saveSearch;
    private TextField searchName;
    private AddressBookApplication app;

    public SearchView(final AddressBookApplication app) {
        this.app = app;

        setCaption("Search contacts");
        setSizeFull();

       /* Use a FormLayout as main layout for this Panel */
        FormLayout formLayout = new FormLayout();
        setLayout(formLayout);

       /* Create UI components */
        tf = new TextField("Search term");
        fieldToSearch = new NativeSelect("Field to search");
        saveSearch = new CheckBox("Save search");
        searchName = new TextField("Search name");
        Button search = new Button("Search");
```

```
                    /* Initialize fieldToSearch */
                    for (int i = 0; i < PersonContainer.NATURAL_COL_ORDER.length; i++) {
                        fieldToSearch.addItem(PersonContainer.NATURAL_COL_ORDER[i]);
                        fieldToSearch.setItemCaption(PersonContainer.NATURAL_COL_ORDER[i],
                              PersonContainer.COL_HEADERS_ENGLISH[i]);
                    }

                    fieldToSearch.setValue("lastName");
                    fieldToSearch.setNullSelectionAllowed(false);

                    /* Initialize save checkbox */
                    saveSearch.setValue(true);

                    /* Add all the created components to the form */
                    addComponent(tf);
                    addComponent(fieldToSearch);
                    addComponent(saveSearch);
                    addComponent(searchName);
                    addComponent(search);
                }
        }
```

All available property ids from the **PersonContainer** are added to the field select in the for-loop. We cannot use the **PersonContainer** as a data source for the field as this would add all **Persons** to the drop-down instead of the properties (headers in the table). The `setItemCaption()` sets what is being displayed to more human-readable form, like the headers in the table.

We want to force the user to select a field to search from so we set *nullSelectionAllowed* to false, otherwise an empty value would be added to the select (and this would be selected by default also). We also make the assumption that the user most of the time wants to save his search so the `saveSearch` checkbox is checked by default (value=true).

Now we can run the program and the layout should appear when clicking on the **Search** button. There is one small problem though, nothing happens when you click on **Search**. So we need to add some logic to the view.

We want to add two things: unchecking the `saveSearch` checkbox should hide the `searchName`-field and clicking on the **Search** button should perform a search.

Hiding the `searchName` field when un-checking the `saveSearch` checkbox requires only a few lines of code:

```
saveSearch.setImmediate(true);
saveSearch.addListener(new ClickListener() {
    public void buttonClick(ClickEvent event) {
        searchName.setVisible(event.getButton().booleanValue());
    }
});
```

First we need to set the `saveSearch` checkbox to immediate mode so we get an event every time the user interacts with it. We then attach a click listener to it so we can react to user checking or un-checking the checkbox. We react to the click event simply by setting the visibility of the `searchName` field to the checkbox status (true if checked, false if unchecked).

Performing the search requires a little more effort. We need to pass the input from the user to the **PersonList** (table) so it can filter the result set and only show matching rows. Additionally we need to store the search if the user has checked the `saveSearch` checkbox.

First we create a simple data class, **SearchFilter**, where we can store the input from the user.

```
package com.itmill.dev.addressbooktutorial.data;

public class SearchFilter {

        private final String term;
        private final Object propertyId;
        private String searchName;

        public SearchFilter(Object propertyId, String searchTerm, String name) {
                this.propertyId = propertyId;
                this.term = searchTerm;
                this.searchName = name;
        }

        // + getters
}
```

We then add a click listener to the save button which will fetch the value of the search field and term, create a **SearchFilter** and pass it on to the main application. If saveSearch has been checked we also tell the application to save the search.

```
search.addListener(new Button.ClickListener() {
        public void buttonClick(ClickEvent event) {
                    performSearch();
        }

});
```

As the code is more than a few lines we add it as a separate method:

```
    private void performSearch() {
        String searchTerm = (String) tf.getValue();
        SearchFilter searchFilter = new SearchFilter(fieldToSearch.getValue(),
                searchTerm, (String) searchName.getValue());
        if (saveSearch.booleanValue()) {
            app.saveSearch(searchFilter);
        }
        app.search(searchFilter);
    }
```

Now we need to add the search and saveSearch logic to the application class.

```
public void search(SearchFilter searchFilter) {
      // clear previous filters
      getDataSource().removeAllContainerFilters();
      // filter contacts with given filter
      getDataSource().addContainerFilter(searchFilter.getPropertyId(),
                  searchFilter.getTerm(), true, false);
      showListView();
}
```

The search method simply removes any previous filters and then adds a container filter to the table data source using the data from the searchFilter argument. The true and false parameters to the addContainerFilter() call turns on ignoreCase and makes it match the term as a sub string (not only as prefix, i.e. it will search for "*term*", not "term*").

The filter also needs to be cleared when the user selects "Show All" in the tree. This can be done in itemClick():

```
public void itemClick(ItemClickEvent event) {
        if (event.getSource() == tree) {
            Object itemId = event.getItemId();
            if (itemId != null) {
                if (itemId == NavigationTree.SHOW_ALL) {
                        // clear previous filters
```

```
                    getDataSource().removeAllContainerFilters();
                    showListView();
                } else if (itemId == NavigationTree.SEARCH) {
                    showSearchView();
                } else if (itemId instanceof SearchFilter) {
                    search((SearchFilter) itemId);
                }
            }
        }
    }
```

The `saveSearch()` method adds a new item to the tree, under the **Search** node and then selects that node:

```
    public void saveSearch(SearchFilter searchFilter) {
        tree.addItem(searchFilter);
        tree.setParent(searchFilter, NavigationTree.SEARCH);
        // mark the saved search as a leaf (cannot have children)
        tree.setChildrenAllowed(searchFilter, false);
        // make sure "Search" is expanded
        tree.expandItem(NavigationTree.SEARCH);
        // select the saved search
        tree.setValue(searchFilter);
    }
```

The `searchFilter` is used as the `itemId` for the tree node. A tree uses the `toString()` method of the `itemId` to determine what to display to the user by default (this can be changed by `setItemCaptionMode()`). We thus need to implement a `toString()` method in our **SearchFilter** class which returns what we want to display to the user:

```
    @Override
    public String toString() {
        return getSearchName();
    }
```

Using the `searchFilter` object as the itemId in the tree makes it easy to add a case to the `itemClick()` handler for the tree so the search is performed every time the saved search is clicked. Make the following addition to `itemClick()` handler in **AddressBookApplication**:

```
public void itemClick(ItemClickEvent event) {
        if(event.getSource() == tree) {
[...]
    } else if (itemId instanceof SearchFilter) {
                        search((SearchFilter) itemId);
    }
[...]
}
```

Now we have created a search view, implemented the search functionality and also support for saving searches.

You can download the Eclipse project package for the end of this step from:

- http://www.itmill.com/download/doc/tutorial/itmill-toolkit-5-tutorial/ab-ch5-ui.zip

Chapter 6. Tuning the user experience

The application we have built so far is a fully functional application, but lacking some of the nice stuff which would make it easier to use. For example emails in table could be links that open the user's default email application. Other examples discussed here are validation of user input in forms and giving proper feedback to users.

6.1. Turning email addresses into links

The email addresses should be links for directly sending an email and we should validate the data entered to avoid unintentional mistakes.

Figure 6.1. Email Adresses as Text

First name	Last name	Email	Phone number	Street Address	Postal Code	City
Lisa	Schneider	lisa.schneider@itmill.com	+358 02 555 7531	561-9262 Iaculis Avenue	69761	Stockholm

The table we have created and populated so far only lists data directly from the datasource. We would, however, want to turn the email addresses into links so the user can directly click on them to send email. We can accomplish this by using the generated column feature of the Table as shown in the code below. By adding a generated column with the same id as an existing column ("email") we override the existing column and display the generated column instead. Using a different id would result in both the plain and the linked email column being shown. The following code should be added to the PersonList constructor before setting the data source.

```
// customize email column to have mailto: links using column generator
    addGeneratedColumn("email", new ColumnGenerator() {
        public Component generateCell(Table source, Object itemId,
                Object columnId) {
            Person p = (Person) itemId;
            Link l = new Link();
            l.setResource(new ExternalResource("mailto:" + p.getEmail()));
            l.setCaption(p.getEmail());
            return l;
        }
    });
```

The **ColumnGenerator** uses a simple interface with only one method, `generateCell()` which is called every time a value for the column is needed. Passed to this method is the *itemId* for the row which in our example is the same as the **Person**. We acquire the email address from the **Person** object and generate a **Link** component with the standard HTML "`mailto:<address>`" target. Returning the **Link** component from the generator will make the **Table** automatically insert it in the correct position (in place of the original, plain text email address).

The table now contains clickable email links:

Figure 6.2. Email Adresses as Links

First name	Last name	Email	Phone number	Street Address	Postal Code	City
Lisa	Schneider	lisa.schneider@itmill.com	+358 02 555 7531	561-9262 Iaculis Avenue	69761	Stockholm

6.2. Notifications

Notifications are a nice way of informing the user what is happening without adding e.g. a Label which requires some space in the view. Let's test the notifications by adding a notification when searching, so the user sees what he searched for. This will be displayed at the same time as the results are displayed. We will add the notification for both normal and saved searches so the correct place for the code is the search method in the application class.

```
getMainWindow().showNotification(
        "Searched for " + searchFilter.getPropertyId() + "=*"
        + searchFilter.getTerm() + "*, found "
        + getDataSource().size() + " item(s).",
        Notification.TYPE_TRAY_NOTIFICATION);
```

Now the user gets a nice little popup in the bottom right corner of the screen which tells what the search criteria was and how many rows matched. *Notification.TYPE_TRAY_NOTIFICATION* places the notification in the lower right corner, feel free to experiment with other types to see what happens.

Figure 6.3. Tray Notification
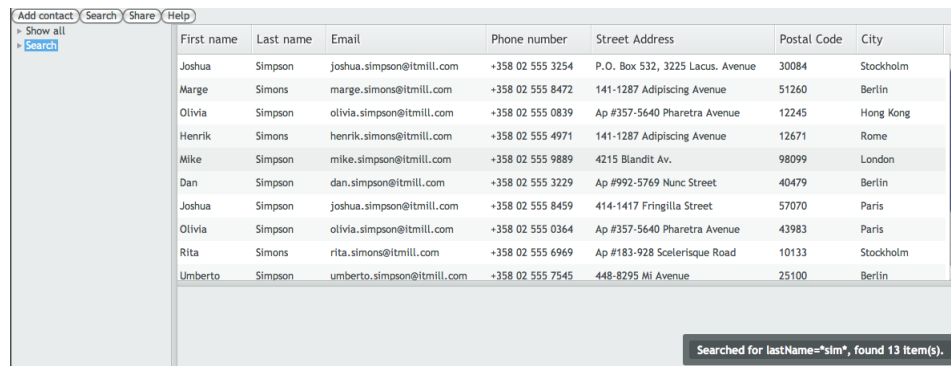
6.3. Using a combo box for fluent city selection

If you have tried the final version, you have probably noted the **ComboBox** component used for selecting a city. Using it for selecting an existing city becomes much more efficient. Replacing the normal text field with a combo box will introduce you to the system used in IT Mill Toolkit for populating a form. The same mechanism can be used in tables if you put the table into "editable" mode . Fields are generated using the strategy pattern by a class implementing the **FieldFactory** interface.

By default the **BaseFieldFactory** class is used and is generally a good base for customizations. The **FieldFactory** interface has several methods which we do not want to implement and **BaseFieldFactory** has a good implementations for these. Later we'll use the same field factory to tune the form fields even more.

We start by adding a **ComboBox** field to our **PersonForm**:

```
private final ComboBox cities = new ComboBox("City");
```

We make adding of new cities possible and populate the combo box with existing cities from our data source in the constructor. We cannot have duplicate itemIds in a **ComboBox** so each city will show up only once. Add the following code to the **PersonForm** constructor.

```
/* Allow the user to enter new cities */
 cities.setNewItemsAllowed(true);
 /* We do not want to use null values */
```

```
cities.setNullSelectionAllowed(false);
/* Add an empty city used for selecting no city */
 cities.addItem("");

/* Populate cities select using the cities in the data container */
 PersonContainer ds = app.getDataSource();
 for (Iterator<Person> it = ds.getItemIds().iterator(); it.hasNext();) {
     String city = (it.next()).getCity();
     cities.addItem(city);
 }
```

Finally we'll make a field factory extending the **BaseFieldFactory**. Using the field factory we can choose what kind of field to use for each property. For the city property we'll return the cities combo box, for all other properties we'll let the **BaseFieldFactory** take care of creating the field.

```
/*
  * Field factory for overriding how the component for city selection is
  * created
  */
 setFieldFactory(new BaseFieldFactory() {

     @Override
     public Field createField(Item item, Object propertyId,
        Component uiContext) {
        if (propertyId.equals("city")) {
           return cities;
        }

        return super.createField(item, propertyId, uiContext);
     }

});
```

6.4. Automatically validate user input

We want to validate two of the fields in the form: the postal code field should contain 5 numbers and not start with a zero, and the email address should have a valid form.

For the postal code we create a **PostalCodeValidator** which implements the **Validator** interface. We implement the two required methods: `isValid()` which just returns true or false depending on if the value is valid and `validate()` which throws an **InvalidValueException** containing a description of the error for invalid values. To avoid duplicating code we use `isValid()` inside the validate function. The actual validation is done by the regular expression match return

```
(String) value).matches("[1-9][0-9]{4}")
```

which returns true only for a 5 digit **String** where the first character is `1-9`.

```
package com.itmill.dev.addressbooktutorial.validators;

import com.itmill.toolkit.data.Validator;


public class PostalCodeValidator implements Validator {
    public boolean isValid(Object value) {
        if (value == null || !(value instanceof String)) {
            return false;
        }
        return ((String) value).matches("[1-9][0-9]{4}");
    }

    public void validate(Object value) throws InvalidValueException {
        if (!isValid(value)) {
```

```
            throw new InvalidValueException(
                    "Postal code must be a five digit number and cannot start with a
zero.");
        }
    }
}
```

We assign the validator to the field in our **FieldFactory** and at the same time make the field required. We make another improvement at the same time by changing the null representation for the field. Instead of the default "`null`" we want to display an empty field before the user has entered any value. The code for the `postalCode` field in the **FieldFactory** is now:

```
/*
 * Field factory for overriding how the component for city selection is
 * created
 */
setFieldFactory(new BaseFieldFactory() {
    @Override
    public Field createField(Item item, Object propertyId,
            Component uiContext) {
        if (propertyId.equals("city")) {
            return cities;
        }
        Field field = super.createField(item, propertyId, uiContext);
        if (propertyId.equals("postalCode")) {
            TextField tf = (TextField) field;
            /*
             * We do not want to display "null" to the user when the
             * field is empty
             */
            tf.setNullRepresentation("");

            /* Add a validator for postalCode and make it required */
            tf.addValidator(new PostalCodeValidator());
            tf.setRequired(true);
        }

        return field;
    }
});
```

The field now contains a red star, telling the user that it must be filled out. It is also initially empty.

Figure 6.4. Validated Field Initially Empty
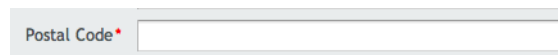
If the user enters an invalid value into the field a red exclamation mark appears and when the user moves the mouse cursor over the field a tooltip appears, containing the error message from the validator.

Figure 6.5. Validated Field with Invalid Value
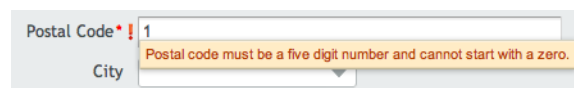
We do the same thing for the email field but use a different validator implementation which checks that the email is of form *@*.*. Implementing the EmailValidator is left here as an exercise for the reader (the implementation can be found in the **EmailValidator** file in the final package).

6.5. Enabling advanced features in a Table

The **Table** component in IT Mill Toolkit is quite powerful. There are several features available that enhance the user experience. Some of these (like sorting) require support from the data source. The best place to enable these features is naturally the constructor in our **PersonList** class.

Sorting is on by default provided the data source supports it. The **BeanItemContainer** which our **Person-Container** is based upon has sorting support for every field whose type implements **Comparable**. Try clicking on column headers to sort the table according to any column. Sorting can be explicitly disabled if necessary.

Other features enhancing the usability of a Table are column reordering and column collapsing. These are disabled by default. Add the following lines to **PersonList** and then try dragging column headers. Also try the menu appearing in the top right corner of the table.

```
setColumnCollapsingAllowed(true);
setColumnReorderingAllowed(true);
```

6.6. Summary

We have in this chapter added some basic usability enhancing features to the Address Book application. One final thing still remains before the application is ready and that is changing the design, or theme as it is called in a Toolkit application. This is done in the next chapter.

You can download the Eclipse project package for the end of this step from:

- http://www.itmill.com/download/doc/tutorial/itmill-toolkit-5-tutorial/ab-ch6-finetuning.zip
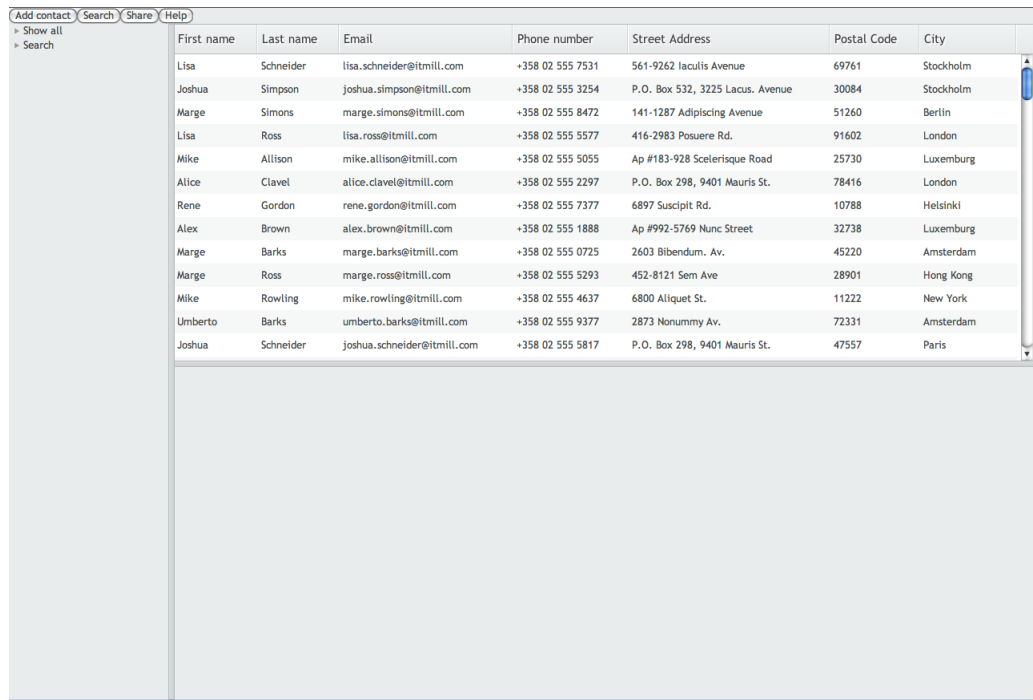
Chapter 7. Building a Simple Theme

A toolkit application always uses a theme which defines the look and feel of the application. If no theme is specified the "`default`" theme, shipped with IT Mill Toolkit, is used and the application looks like this.

Figure 7.1. Layout with Default Theme

We have now arrived at the stage where we want the application to look a bit more like the marketing department has visioned it. They have decided to brand our application "Contacts!" and want a few images added and want to change the color theme.

7.1. Using a Custom Theme

We want to change the looks of the application and need to start by creating our own theme. We decide to name our theme "`contacts`". Creating a new theme is simple, we will extend the default theme to keep the current look and then make some customizations to change the looks to parts of the application. Creating a new theme can be summarized in the following few steps:

- Create a theme folder "`contacts`" in `WebContent/ITMILL/themes`

- Create a `styles.css` inside the "`contacts`" folder. This defines all the css used in our theme. To include the default theme we add

  ```
  @import url(../default/styles.css);
  ```

  at the beginning of the `styles.css`

- Tell the application to use our custom theme. This is done by calling `setTheme()` when creating our application, so we add the code

```
setTheme("contacts")
```

to the `buildMainLayout()` method in **AddressBookApplication**.

The application now uses our custom theme (but you won't notice it as we have not made any changes yet). If you are completely new to CSS (Cascading Style Sheets), please have a look at e.g. http://www.w3schools.com/css/.

7.2. Adding some space around the components

In the original layout everything is pretty much packed together and a little extra space here and there wouldn't hurt. We can add margins to layouts using CSS (can select the size of the margin) or by using the `setMargin()` method in the code (uses a default size which can be overridden in CSS). Here we have chosen to use the default margin sizes and just enable the margins in the code. We add margins and spacing to the buttons in the toolbar by adding the following to the application classes `createToolbar()` method:

```
lo.setMargin(true);
lo.setSpacing(true);
```
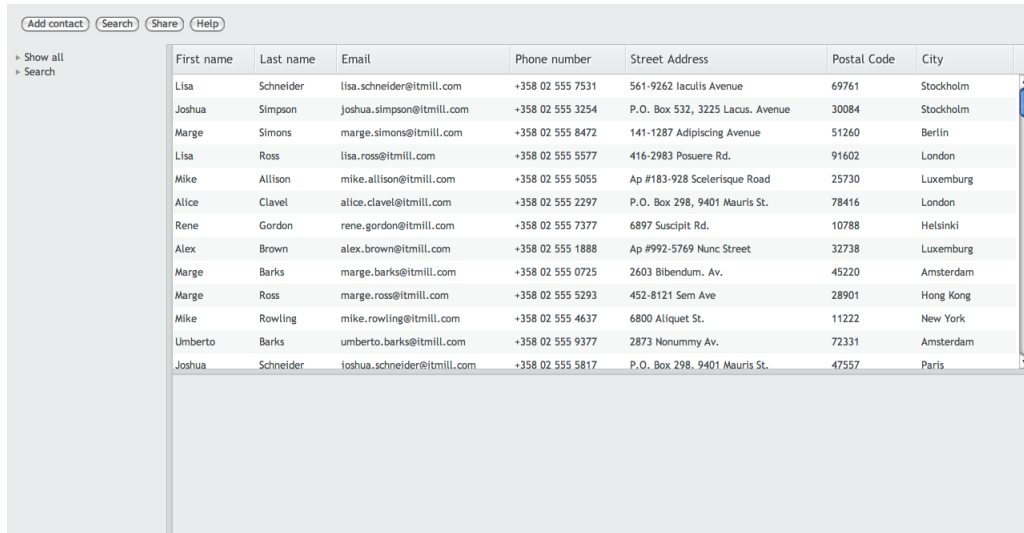
We also want a little space around the tree which we can get by adding two lines of CSS to our `styles.css`, which now looks like the following:

```
@import url(../default/styles.css);

.i-tree {
    padding-top: 8px;
    padding-left: 4px;
}
```

Now the application already looks a little less packed:

Figure 7.2. Layout with Spacing

Did it not work and everything disappeared or just looks broken? Make sure your theme name is correct and that the @import line is present in your `styles.css`.

7.3. Images and icons

We are now ready to add some icons to the navigation buttons in the header (**Add contact**, **Search**, etc.). We will add the icon files to an `"icons"` folder below our theme directory (`"contacts"`) and then use them for the buttons. The icon files are copied from the default theme included with IT Mill Toolkit and are also included in the downloadable package for this step (see Section 7.4, "Final version" for the download link). To add an icon to a button we use the `setIcon()` method:

```
search.setIcon(new ThemeResource("icons/32/folder-add.png"));
```

The **ThemeResource** refers to a file found in the theme directory. This code goes into **AddressBookApplic- ation**.`createToolbar()` where the buttons are created. We add icons for the other buttons in the same way:

```
share.setIcon(new ThemeResource("icons/32/users.png"));
help.setIcon(new ThemeResource("icons/32/help.png"));
newContact.setIcon(new ThemeResource("icons/32/document-add.png"));
```

We also want to add a logo for the application in the right hand part of the header. There is no need for this to be a **Button** so we add an **Embedded** component to display the logo instead. The logo can be found in the "images" folder in the theme directory so we can access it by using a **ThemeResource**, the same way we accessed the icons.

```
lo.setWidth("100%");

Embedded em = new Embedded("", new ThemeResource("images/logo.png"));
lo.addComponent(em);
lo.setComponentAlignment(em, Alignment.MIDDLE_RIGHT);
lo.setExpandRatio(em, 1);
```

We need a little extra code to get the logo where we want it. First of all we want the toolbar to use all available space horizontally so we set its width to `100%`. We set the alignment for the logo to the right so it is placed at the right hand edge of the application. We also set the expand ratio for the component so that all extra space available in the layout will be assigned to this component. The result is that the navigation buttons are placed at the left side, the logo at the right side and all extra, empty space is placed between the logo and the buttons. The application now looks like the following:

Figure 7.3. Themeing Icons
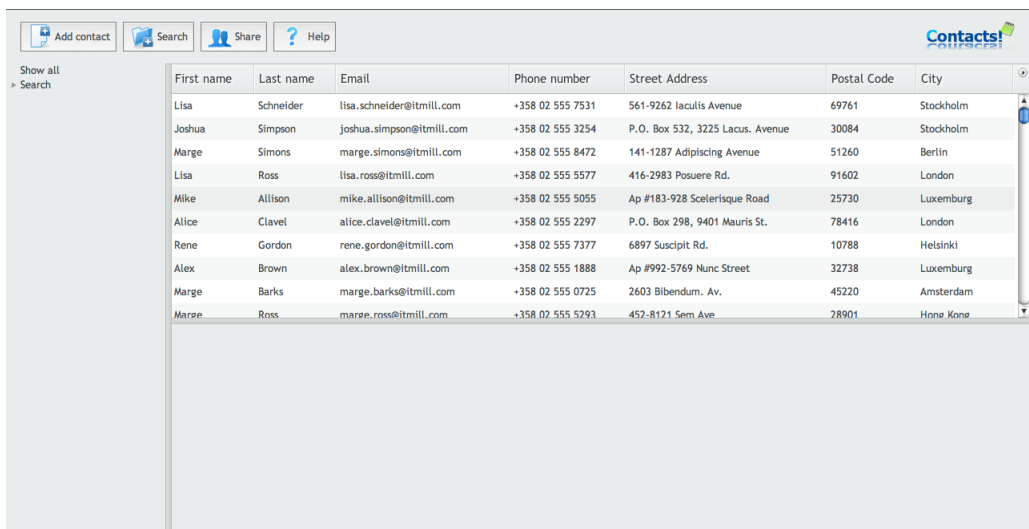
Even though we got our icons in place we are still not entirely happy about the way the buttons look. We would like to take the borders away and place the text below the image. We also want the text to be a little smaller and have a small shadow.

We accomplish this by first assigning a style name to the toolbar using:

```
lo.setStyleName("toolbar");
```

We can now use CSS to style the buttons inside the toolbar:

```
.toolbar .i-button {
     display: block;
     width: 65px;
     height: 55px;
     background: transparent;
     border: none;
     text-align: center;
}

.toolbar .i-button img {
     display: block;
     margin-left: auto;
     margin-right: auto;
     margin-bottom: 5px;
}

.toolbar .i-button span {
     font-size: x-small;
     text-shadow: #fafafa 1px 1px 0;
}
```

We include .toolbar in the CSS file to make sure that we only change the looks of the buttons in the toolbar (this refers to the styleName that we set earlier).

Figure 7.4. Themeing Toolbar
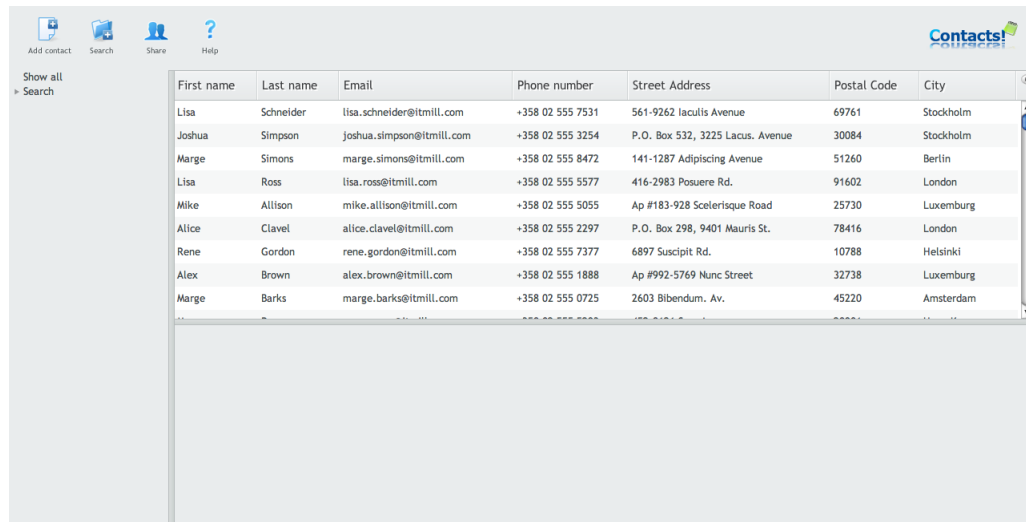
Using the above CSS the application now looks like the following:

Next we add a background to the header and change some background colors and the default font. In order for our background change to affect all views we need to add style names to all of our views:

```
addStyleName("view");
```

The same style name can be added to both **ListView** and **SearchView**. The CSS we use is:

```
.i-app {
    background: #d0e2ec;
    font-family: "Lucida Grande", Helvetica, Arial, sans-serif;
    color: #222;
}

.toolbar {
    background: #ccc url(images/gradient.png) repeat-x bottom left;
}

.i-panel-caption-view {
    color: #004b98;
}

.view {
    background: white;
}
```
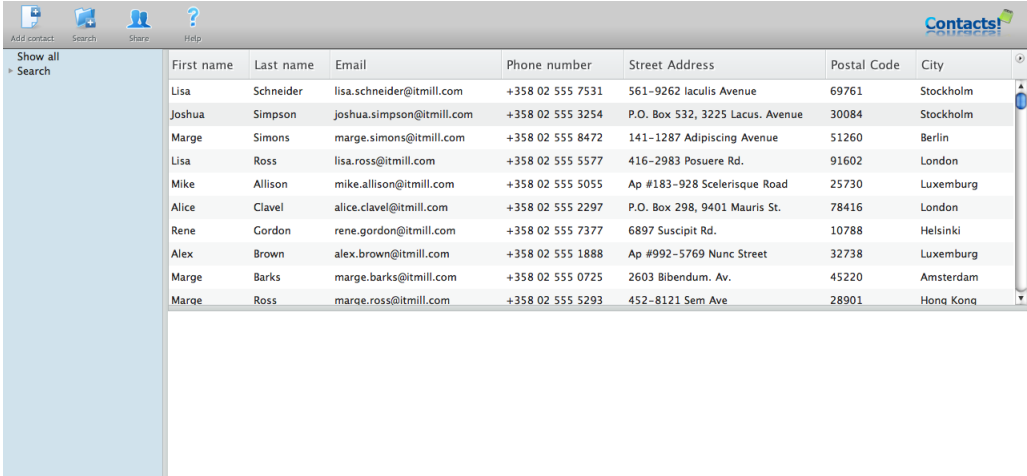
And we get:

Figure 7.5. Themeing View

We still want to theme the table a bit to use a smaller font and some other colors so we use the following CSS:

```
/* Theme table to look bit lighter */
.i-table-header-wrap {
    height: 20px;
    border: none;
    border-bottom: 1px solid #555;
    background: transparent url(images/table-header-bg.png) repeat-x;
}

.i-table-caption-container {
    font-size: 11px;
    color: #000;
    font-weight: bold;
    text-shadow: #fff 0 1px 0;
    padding-top: 1px;
}

.i-table-body {
    border: none;
```

```
}

.i-table-row-odd {
    background: #f1f5fa;
}

.i-table-row:hover {
    background: #fff;
}

.i-table-row-odd:hover {
    background: #f1f5fa;
}

.i-table .i-selected {
    background: #3d80df;
}

.i-table-cell-content {
    padding: 2px 0 2px 3px;
    line-height: normal;
    font-size: 85%;
}
```
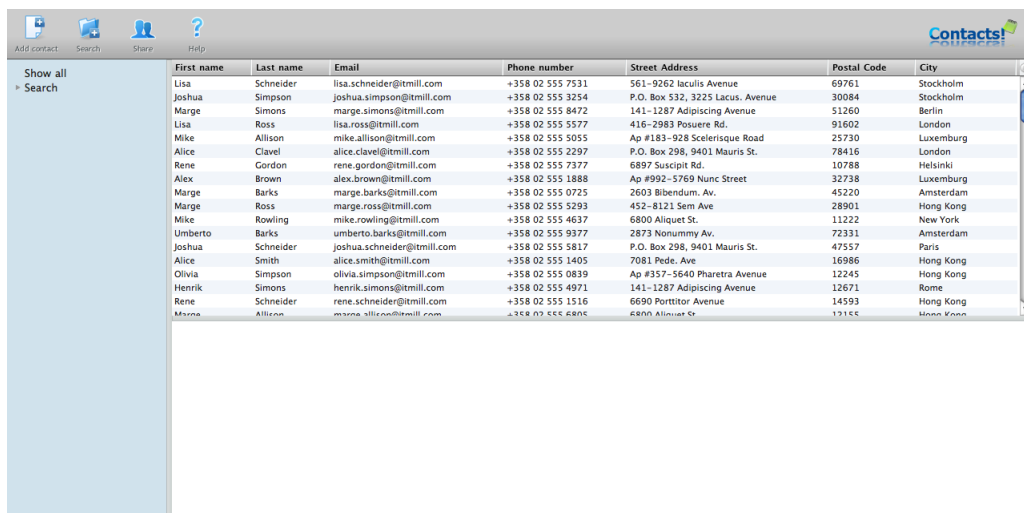
And get our final result which both we and our marketing department are happy with:

Figure 7.6. Final Graphical Design

7.4. Final version

We have made a final version of the application which includes a few more features including:

- Equal width for all form fields

- Notifications if necessary parameters are missing from the search view

- Ability to close the "share"-popup using the **OK** button

You can download the Eclipse project package for this final step from:

- http://www.itmill.com/download/doc/tutorial/itmill-toolkit-5-tutorial/ab-final.zip