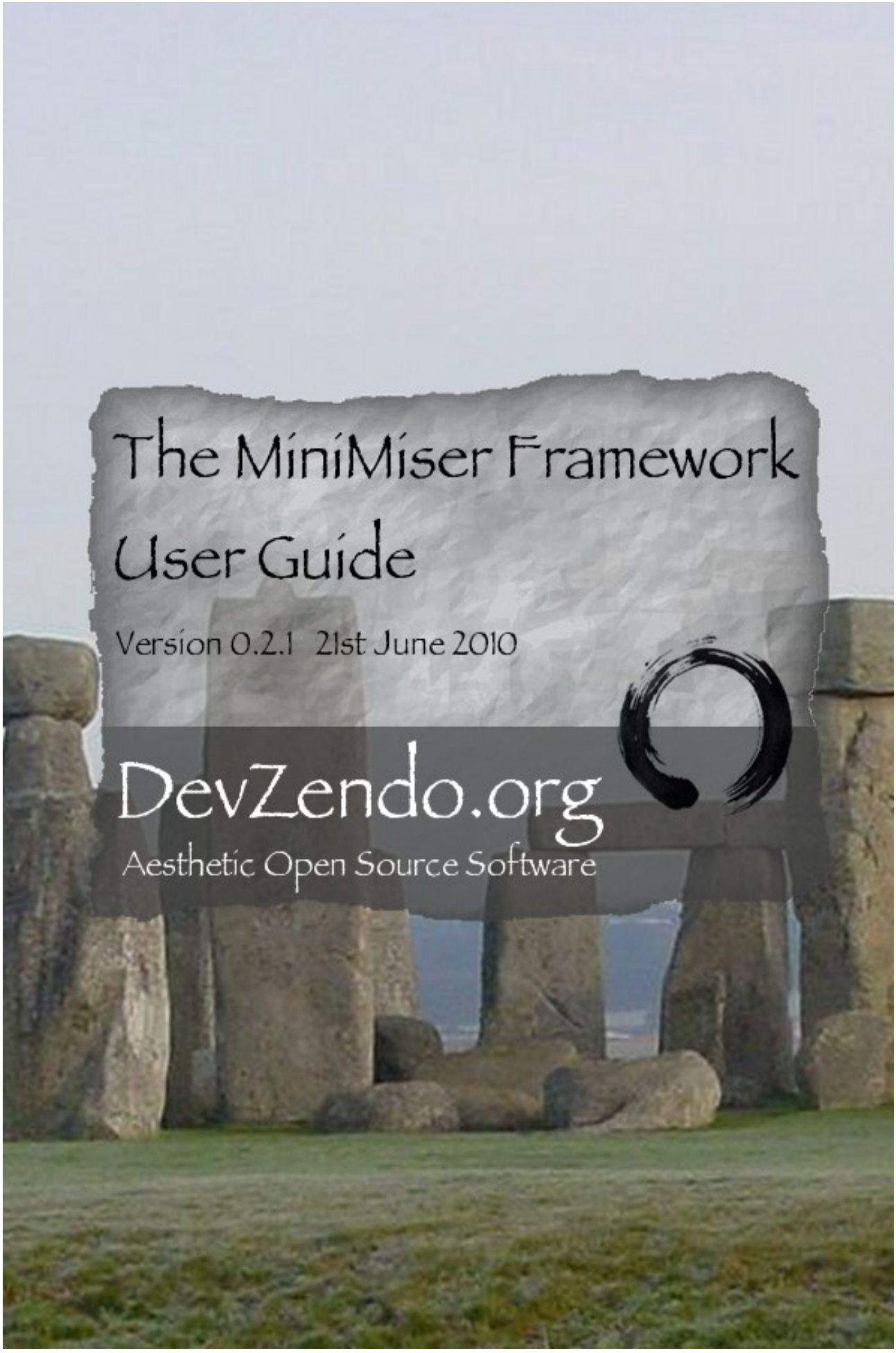

The MiniMiser Framework
User Guide
Version 0.2.1 21st June 2010
DevZendo.org
Aesthetic Open Source Software

MiniMiser Framework User Guide

http://devzendo.org/content/minimiser-framework

Table of Contents

Part I - Introduction 5
About the MiniMiser framework 5
About this document 5
Typographical conventions 5
Intended readership 6
Framework overview 6
Database storage 6
Multiple open databases 7
Session storage 7
Look and Feel 7
Menu building 7
Multiple views into databases 7
Configuration and preferences 8
Plugin management 8
Update notification 8
Messaging 9
Background processing 9
What MiniMiser is not 9
A note on backwards compatibility 10
For more information 10
Part II – Getting Started 11
Setting up your development environment 11
Eclipse and Eclipse Plugins 11
Maven settings 11
Introducing the sample application 12
Creating your application project in Eclipse 13
Developing the sample application 15
Creating the application plugin 15
Creating the plugin descriptor 16
Overview of the project directory structure 17
Running your plugin with MiniMiser 17
Creating the database schema 18
Using the SpringLoader 22
Creating the Data Access Object layer 22
The domain layer and the DAO pattern 22
The sample application DAO layer 25
Making use of the DAO layer with the framework 27
Driving the development of the DAO layer with tests 27
Creating the User Interface 30
Customising the menu 30
Creating View menu items 30
Creating Tabs 30
Part III – MiniMiser Framework Reference 31
Programming languages 31
Plugin architecture 31

Scanning for plugins.properties files..... 31
Instantiating plugin classes..... 31
Initialising plugin objects..... 31
ApplicationPlugin Resources..... 32
Separation of concerns: Developing your plugin with Facades..... 32
Persistence..... 34
Access mechanisms..... 34
Database schema for version control..... 34
Database creation facade..... 35
Database opening facade..... 35
Database closing facade..... 36
Database migration facade..... 36
File storage alongside databases..... 38
Presentation..... 38
Database creation wizard pages facade..... 38
Menu system..... 38
The ApplicationMenu object(s)..... 39
The TabIdentifier object..... 40
The OpenDatabaseList object..... 40
The DatabaseDescriptor object..... 41
The MenuFacade object..... 41
View system..... 41
Tab lifecycle management..... 41
Status Bar..... 43
Dialogs..... 43
The About Dialog..... 43
The Welcome / What's New Dialog..... 43
The Problem Reporter Dialog..... 44
Indicating long-running tasks with the CursorManager..... 45
Messaging..... 45
Messages with “Don't Show This Again” checkboxes..... 45
Options or Preferences..... 45
Miscellaneous..... 45
Logging..... 45
Lifecycle management..... 46
Event notification..... 46
Database event notifications..... 46
Tab event notifications..... 46
Upgrade event notifications..... 46
The startup queue..... 46
Background processing with WorkerPool and DelayedExecutor..... 46
The change log..... 46
The change log markup syntax..... 47
Example..... 47
Structure..... 47
Header line structure..... 47
Optional information line structure..... 48
The update notification system..... 48

Overview 48
Update availability notification 48
Privacy concerns 49
Making use of an update site 49
Periodic update availability checks 49
User-driven update availability checks 50
Resources 50
Deployment and Packaging 50
Overview 50
Structuring cross-platform application projects 50
Using the CrossPlatformLauncherPlugin 51
Mac OS X structure 51
Linux structure 51
Windows structure 52

Part I - Introduction

About the MiniMiser framework

The MiniMiser framework is a small desktop application development framework intended to provide common functionality to programs that use an embedded SQL database as their persistence mechanism, with a rich client graphical user interface.

I started developing it in April 2008 as part of a home financial management product, similar in scope to Microsoft Money, Quicken, etc., as the commercial product I was using on Windows XP had been dropped by its developers, and was no longer supported. It was also the only software for which I needed Windows XP.

I gave the program the code-name “MiniMiser”, since it was intended to be small, and to help one to be frugal with one's finances.

It is open source – to prevent the 'lack of developers / unsupported / closed system' problem occurring again, and also multi-platform – running on Linux, Mac OS X, and Windows.

An aim is to make it user-friendly, and understandable by capable but nontechnical users. Design decisions are often guided by the thought “What would Apple do?”. Where possible, I follow the Apple Human Interface Guidelines, and strive for a simple – but not simplistic – interface. The software runs on Mac OS X, Ubuntu Linux and also Microsoft Windows.

I also hope that an open source development community might arise to make use of it, and to further its development. To produce a product that is cross-platform with many developers implies the use of Java as primary development language, although as the project has progressed, the possibility of developing plugins in alternative JVM-based languages has also been investigated.

I didn't intend to write a framework – it was just going to be this one product. However, I found myself wanting to reuse parts of the code in other projects, so started splitting it into a general purpose framework - or substrate - upon which to build similar programs, including the home finances product. That product has received little attention as of yet, due to the scope of the foundations of the framework.

The product was renamed with the code-name “BeanMinder” (which is awful, admittedly), and the framework itself became “MiniMiser”, as it is small, and minimises the amount of work you have to do to build products.

About this document

This document illustrates how to write desktop applications that use the MiniMiser framework. It starts with a tutorial, and then presents reference material that describes the framework, and what can be achieved with it.

Typographical conventions

Example code is shown in a monospaced font:

```
public class MyApplication extends AbstractPlugin implements ApplicationPlugin {
```

Long items of text that should be kept on a single line, but that would exceed the margins in this document are shown with a ’ character at the line break. Do not insert a line break in your actual code:

```
class="org.devzendo.beanminder.plugin.facade.newdatabasecreation.’
    BeanMinderNewDatabaseCreationFacade" />
```

Intended readership

If you want to write small, database-backed, rich client GUI applications with a minimum of fuss, and haven't got time (or tolerance of pain/frustration) to master the large, more enterprisey frameworks out there, then this document should help you get started with your application.

You will need:

- A reasonable knowledge of Java 5.
- A suitable development environment (the framework and sample applications are all developed using Eclipse 3.3/3.4/3.5)
- Some knowledge of Maven 2 (we use version 2.0.9 and up; you will find the Maven Integration plugin m2eclipse version 0.9.8 invaluable). “Maven: The Definitive Guide” from O'Reilly is an excellent book on the subject. It is also available online, from its authors: see http://www.sonatype.com for details.
- Some knowledge of Spring – we use it as a dependency injection container, and use Spring JDBC Templates for data access. “Spring in Action” by Manning Press is an excellent book on the subject.
- Some knowledge of design patterns would be useful; we don't over-use them.
- Some knowledge of JUnit and EasyMock, and how they are used in test-driven design and development (TDD) would be of great benefit in creating a high-quality, reliable, well-designed software.

A later section of this document will detail how to set up your development environment in order to develop MiniMiser applications.

A separate document details additional changes you would need in order to join the development of the framework itself.

Framework overview

The framework provides the following features, most of which can be extended / customised by your own applications:

Database storage

MiniMiser apps store their persistent data in small, embedded relational databases, using the H2 SQL database engine. All files relating to a single database are stored in their own directory, which is given the name of the database. Databases can therefore be managed on disk by moving folders.[1]

1 Note that if a database folder is moved, the 'recent files list' will not be able to find it. Also, any databases that were open on exit will be reloaded on startup – if any have moved, they will not be opened. The user must re-open them

H2 was chosen as the persistence engine due to its heritage and maturity, coming from the same developer as the earlier Hypersonic database. It also permits databases to be completely encrypted using the AES-256 encryption algorithm – an essential feature given the sensitive nature of the financial information I am hoping to store in my application. The user chooses whether the database is to be encrypted at creation time, and will be queried for the password upon subsequently re-opening the database (manually, or automatically, on startup).

Databases always contain the application's version number. This allows databases created by older versions of the software to be automatically updated to the latest schema when they are opened by newer versions of the software. The framework provides mechanisms by which you can provide upgrade steps to aid this process.

The framework itself uses Spring JDBC Templates to simplify access to the database via JDBC. The architecture so far assumes you are using Spring JDBC Templates to access your data using an implementation of the Data Access Object pattern – the essentials of this are covered later, but see Martin Fowler's Patterns of Enterprise Application Architecture ('Mapper' pattern), or Eric Evans' Domain Driven Design ('Repository' pattern) for further details.

This does not preclude the use of Hibernate to provide higher-level access using Object-Relational mapping; this has not yet been explored. Similarly for iBatis SQL maps. If you are interested in pursuing either of these strategies for data access, please contact us!

Multiple open databases

Multiple databases can be open simultaneously. The user can switch between them at will. In presenting multiple open databases, MiniMiser does not use a Multiple Document Interface approach. There is only one MiniMiser main window, whose contents change completely when a database switch is effected. This does create a restriction in that one cannot see two open databases simultaneously. This may be changed in the future.

Session storage

Much of the users' current 'position state' is persisted and restored on startup: the set of open databases; the current database; the open views into each database; the current view in each database; the main window size and position. Restoring this session state manually upon restarts of the software is deemed too demanding a cognitive load on the user, and something they should not have to consider.

Look and Feel

The framework uses the Quaqua look and feel on Mac OS X, which provides a better Mac-like UI than the standard Apple look, especially for file choosers. This requires the use of Quaqua's native libraries, which should be packaged alongside the Quaqua jar (with the rest of the MiniMiser dependencies). This will be covered fully in the sections on launching and packaging. At the time of writing, Quaqua is not available from the central Maven repository. It is, however packaged into a jar and zip of native libraries, and provided in the DevZendo.org Maven repository.

On Windows and Linux, the framework uses the JGoodies Plastic XP look and feel.

---

manually, in order for the framework to add them to the 'recent files' and 'open files' lists.

Both these libraries are declared as dependencies of the framework; you may filter them out as necessary if only developing for specific platforms.

Menu building

The framework builds the application's menu for you, giving you a File, View, Tools, Window and Help menu all with the functionality you'd expect for opening/closing databases, switching between multiple open databases, opening and closing different views/screens. Your application may also customise the menu bar with its own menus.

Multiple views into databases

The framework also makes several assumptions about the presentation of the various views into the database, and gives some small opportunity for customisation:

The presentation of the various 'screens' of the application is to be split into several 'views', all of which are available from the View Menu. There is always an Overview View available, and an SQL Console View[2]. Views can be opened or closed. However, the Overview View is not closable – there must always be at least one view open – having an empty screen after you have opened a database does not yield a positive user experience. Each view is presented as a tab in the application main window. This does create a restriction in that one cannot see two views simultaneously. This may be changed in the future.

Plugins contribute to the set of views by declaring their names, and, on demand, creating their panels which the framework shows when the relevant tab is selected in the main application window.

Configuration and preferences

There are common dialogs for editing user preferences, backed by a preferences file. This file is held in a directory under the user's home directory, and is in Windows .INI format.

As with the view panels, plugins can contribute panels to the preferences dialog.

Plugin management

Multiple plugins can customise the framework to provide a single deployable application. To deliver an end-user application, you must provide exactly one main application plugin that provides the main domain-specific functionality, augmented by other non-application plugins.

As is the case with the framework as a whole, plugins can provide common functionality – for example, the storage and presentation of 'tags' for categorising your application objects, or the ability to add notes to your application and its objects.

Plugins can customise the runtime environment by hooking into the Spring dependency injection container. They can provide their own application context files, and reference beans declared in the framework's context. They can also override beans declared by the framework, although this facility

2 The SQL Console View is a developer debug/diagnostic console in which SQL / DDL / DML commands may be issued upon the currently active database. It also provides tabular views of table SELECTs, and a rudimentary command history, similar to UNIX shells. You must enable an option in the application Preferences dialog, in order to see the SQL view on the View Menu, and to invoke it.

is discouraged.

The methods provided in the application's main plugin class are called by the framework during a typical session – throughout the framework startup and shutdown sequence, when creating, opening and closing databases, switching views, and to present plugin-specific views.

Application plugins may also provide an update site URL, from which notifications of new releases are obtained. They provide text for the 'About' / 'Welcome' dialogs and the license text.

Plugins are currently written in Java, although investigations have started into using JRuby instead.

Update notification

There is an update notification mechanism: the application plugin can register an update site URL. The framework will regularly check this URL to determine the latest version of the plugin, and, if this is different to the installed version, will display a message indicating that there is an update available, and also display a list of the changes in the new version.

Note that only application plugins are allowed to register an update site URL: normal plugins cannot. This is because the end user doesn't usually care about the internals of your application – they're under the hood, and in some cases, shared by multiple applications[3] - and also bundled into a given application. If an end-user was to receive an update notification for such an internal plugin, it is of no interest to her – she cannot upgrade just the internal plugin to take advantage of the new features. New versions of plugins are of interest to application plugin providers, who would bundle the new internal plugin with a new version of the application, and released an update notification for the application itself.

This update notification system does not constitute a software update mechanism, as found for example in Firefox, where new versions are automatically downloaded and installed. It's merely notification. The operating system upon which the application runs *should* have the notification and update mechanism as part of its capabilities. The framework provides a notification mechanism in case you are running on an OS that does not yet provide these features. On Windows and Mac OS X, you have to manually download new version of the program and install them yourself. On Linux, you would update the application from the package management system (e.g. apt-get / aptitude / Synaptic on Ubuntu).

The framework prompts the user upon initial installation to ask whether they would like to be notified of updates. No calls to update sites will be made unless the user gives consent to this.

Update availability checks can be triggered manually; they are also performed once per day automatically (if permission has been given).

3 For example, the tags and notes plugins.

Messaging

Application code can always display a message box to the user. However, unless the user is expecting the appearance of a dialog, such messages can be startling, and disrupt the user's 'flow'. The framework provides a system whereby lower-priority messages can be directed to the user via a queue. If any messages are placed in the queue, a button appears at the bottom of the application main window, indicating that there are messages waiting to be read. Clicking the button – at the users' leisure – displays the message queue window. The user can then go through the list of messages waiting.

Messages can be presented with a 'Don't show this again' checkbox. If such a message might be annoying upon subsequent display, it can be turned off.

More complex messages can be created, e.g. the request for the user's permission to contact update sites requires phrasing in a different way than the 'Don't show this again' mechanism allows.

Background processing

The framework provides a worker pool for background tasks, and also a time-delay task system, which can be used for tasks such as input field autocompletion based on database lookups, triggered some milliseconds after the user has finished typing.

The bottom of the main window holds a status bar, whose progress bar can be used to indicate progress of background tasks.

What MiniMiser is not

MiniMiser is not:

- A tool for end-users to rapidly design their databases using graphical database schema design tools, or to develop entry forms using graphical form designers, with a built-in easy-to-use integration language. That's Microsoft Access, or possibly OpenOffice.org Base.
- A stand-alone database. That's Oracle, Microsoft SQL Server, MySQL, PostgreSQL, DB2, Ingres, Sybase, etc.
- An ultra-rapid development system for database-backed web applications. That's Ruby on Rails, or Grails/Groovy.

This isn't to say that we won't take on some of the features you might find in the above list of excellent products in the future; we're just focussed on small, desktop, embedded, rich client products at the moment.

A note on backwards compatibility

MiniMiser is developed using an agile, test-driven approach. Refactoring occurs whenever it is needed, and this may cause incompatibility between plugin code, and the main framework. I do not provide masses of backwards-compatibility 'shims' or other devices in order that older plugins will run in five years time. Providing these would add a large measure of unnecessary complexity to the code, and is orthogonal to clean design. Plugin developers are expected to be active on the mailing list, follow deprecation notices, and adjust their code accordingly. Code will be marked as deprecated as necessary, and in these cases, I will try to provide at least one release with such

deprecation, to give plugin developers time to make the necessary adjustments.

For more information

If, after reading this document and working through the examples herein, you find the framework useful, please consider joining the development mailing list. This list carries discussion of both the usage of the framework, and of the development of the framework itself.

Consult http://devzendo.org/content/minimiser-framework for details of the mailing list, how to subscribe/unsubscribe, how to download the source for the framework, and how to contribute.

Part II – Getting Started

This section describes setting up a development environment using MiniMiser to build a small database-backed application, illustrating some of the concepts and mechanisms you need to provide.

Other that specific configuration necessary for using MiniMiser, this guide does not cover the installation or setting up of your development environment or its requisite plugins.

Some 'tricky' aspects of new project configuration, and project launching are covered. A recommended process for developing MiniMiser projects using Test-Driven Development (TDD) is illustrated.

Setting up your development environment

Eclipse and Eclipse Plugins

I assume you're developing with Eclipse 3.5 (3.2, 3.3, 3.4 and 3.5 have all been used successfully).

You will also benefit from version 0.9.8 of the m2eclipse Maven Integration for Eclipse plugin (0.9.3 and above have been used with varying degrees of success) – adapt as necessary for other development environments. Build options and target paths may be different for other versions of the m2eclipse plugin: YMMV.

The m2eclipse plugin update site is at http://m2eclipse.sonatype.org/update/

It is possible to develop MiniMiser applications without Maven, but it is an order of magnitude easier. The .jar files that comprise the framework can be downloaded from our Maven 2 repository manually at http://devzendo-org-repo.googlecode.com/svn/trunk/releases/org/devzendo/ but you'll also have to download all their dependencies (for which, consult the pom.xml files found in the repository). The framework is not released in any other form - there's no convenient 'download' link on the project home page.

If you have not used Maven before, I recommend the O'Reilly book 'Maven: The Definitive Guide'.

I recommend using the Checkstyle plugin, version 4.4.x to perform sanity checks on your code, and adherance to your coding conventions. (If you wish to contribute to the MiniMiser project, the core plugins, or my MiniMiser applications, adherance to my coding style/conventions is highly advised. The DevZendo.org coding standards are available from our source repositories, see http://devzendo.org/content/source-repositories for details)

The Checkstyle plugin update site is at http://eclipse-cs.sf.net/update

Maven settings

MiniMiser and its associated projects are not (yet) distributed via the central Maven repositories. They are currently distributed from a custom Maven repository. To allow easy access to the MiniMiser framework archives, add the following to your settings.xml file. This will be found in

the following typical locations:

Linux: /home/username/.m2/settings.xml

Mac OS X: /Users/username/.m2/settings.xml

Windows: C:\Documents and Settings\username\.m2\settings.xml

The file should be modified so that it includes the following repository details:

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
                      http://maven.apache.org/xsd/settings-1.0.0.xsd">
  ...
  <activeProfiles>
    <activeProfile>default</activeProfile>
  </activeProfiles>
  <profiles>
    <profile>
      <id>default</id>
      <repositories>
        <repository>
          <id>devzendo-org-repository-releases</id>
          <name>DevZendo.org Maven2 releases Repository on Google Code</name>
          <url>http://devzendo-org-repo.googlecode.com/svn/trunk/releases</url>
          <snapshots>
              <enabled>false</enabled>
          </snapshots>
          <releases>
              <enabled>true</enabled>
          </releases>
        </repository>
        <repository>
          <id>devzendo-org-repository-snapshots</id>
          <name>DevZendo.org Maven2 Snapshots Repository on Google Code</name>
          <url>http://devzendo-org-repo.googlecode.com/svn/trunk/snapshots</url>
          <snapshots>
              <enabled>true</enabled>
          </snapshots>
          <releases>
              <enabled>false</enabled>
          </releases>
        </repository>
      </repositories>
    </profile>
  </profiles>
</settings>
```

Introducing the sample application

We are now ready to create a project, containing an application plugin, and run it within MiniMiser. Before launching into the specifics of how we do it, here is an outline of the sample application we'll be building.

BeanMinder is a very simple home finances program that allows you to record credit or debit transactions for a number of accounts. Accounts are held with a certain bank, and have an account code. Credit or debit transactions are made on a certain date, and are later reconciled with your bank statements.

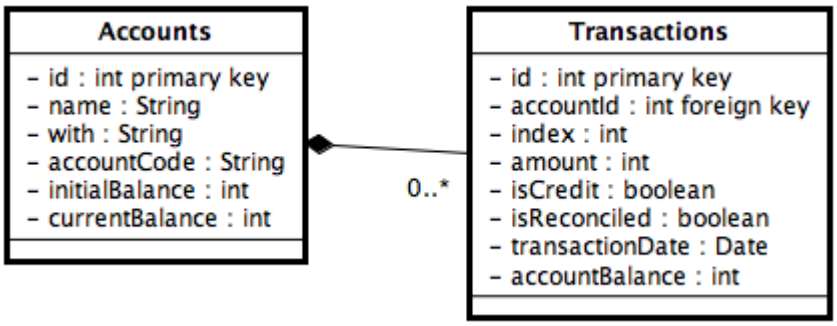
There are two views:

Accounts: lists the accounts and their initial/current balances, from where they can be edited or deleted (with cascading deletion of all associated transactions), and from where a new account can be added. With an account selected, you can switch to the Transactions view.

Transactions: lists the transactions in the chosen account, which can be edited or deleted, and from where new transactions can be added.

Although this is limited, it will illustrate several aspects of MiniMiser : schema creation, data access objects, menus, views and background processing. Admittedly, this is a very simple application, but we'll refine as we work through this guide.

Here's the database schema that the main application plugin will create initially:



The design reasoning behind this schema will be explained in a short while.

MiniMiser doesn't enforce *convention over configuration*, as Ruby on Rails does for table names, but it's a good idea, so I use it here. If you haven't used Rails, the idea is that table names are plural, whereas the objects that you use that represent the rows of the tables are singular. We'll see these objects later, when I discuss the Data Access Object layer.

Armed with this, let's get creating...

Creating your application project in Eclipse

For a Java-based MiniMiser plugin, your project should contain the following standard Maven / m2eclipse source folders:

src/main/java, output to target/classes

src/test/java, output to target/test-classes

src/main/resources, excluding **, output to target/classes (i.e. use directly, don't copy)

src/test/resources, excluding **, output to target/test-classes

It should contain a pom.xml, which should contain a dependency on the framework, and on the artifacts we'll use for testing – JUnit and EasyMock..

To create this in Eclipse, from the Package Explorer, choose New > Other..., then Maven Project, then check the 'Create a simple project (skip archetype selection)' checkbox, then enter something suitable for the Group Id (I chose 'org.devzendo'), and 'BeanMinder' for the Artifact Id.

Once the project is created, you should see the standard Maven source directories as listed above, and the pom.xml. Open the pom.xml, and insert the dependencies on the MiniMiser framework and its test code, JUnit and EasyMock as shown below.

```
<?xml version="1.0" encoding="UTF-8"?>

<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.devzendo</groupId>
  <artifactId>BeanMinder</artifactId>
  <name></name>
  <version>0.1.0-SNAPSHOT</version>
  <description></description>
  <url>http://devzendo.org/content/beanminder/</url>

  <dependencies>
```

```
    <dependency>
      <groupId>org.devzendo</groupId>
      <artifactId>MiniMiser</artifactId>
      <version>0.1.0</version>
    </dependency>
    <dependency>
      <groupId>org.devzendo</groupId>
      <artifactId>MiniMiser</artifactId>
      <version>0.1.0</version>
      <classifier>tests</classifier>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.4</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.easymock</groupId>
      <artifactId>easymock</artifactId>
      <version>2.3</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.easymock</groupId>
      <artifactId>easymockclassextension</artifactId>
      <version>2.3</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

If developing for Mac OS X, add the dependency to the Quaqua native libraries:

```
    <dependency>
      <groupId>ch.randelshofer</groupId>
      <artifactId>libquaqua</artifactId>
      <version>6.5</version>
      <type>zip</type>
    </dependency>
```

There are two ways to make use of these native libraries. If using the CrossPlatformLauncherPlugin, you can also use the Maven Assembly Plugin to unpack them into the right directory in your launcher structure (target/macosx/BeanMinder.app/Contents/Resources/Java/lib). Otherwise, after Maven has downloaded the zip for this dependency, unpack it into some temporary directory.

Once you have added the dependencies and saved the pom.xml, expanding the Maven Dependencies element in the project's tree in Package Explorer should show the dependency on the MiniMiser framework, and several other transitively-dependency artifacts that the framework depends on: log4j, spring, h2, forms, looks, binding, quaqua, wizard, jcalendar, various commons-* artifacts, jruby, asm, jdom, junit and easymock.

You should also consider using Checkstyle to validate your code against a coding standard – by default, it would use Sun's own coding standard; the MiniMiser project uses its own, which is slightly different, and performs several metric analyses on the code being checked. All the sample code in this guide uses that style. To enable CheckStyle for the sample project, right-click on it in Package Explorer, then choose Checkstyle / Activate Checkstyle.

You should also ensure than your project is set to use Java 1.5. Right-click on the project in Package Explorer, then choose Properties. In the properties window, choose Java Compiler, then choose 1.5

(or 1.6) under Compiler compliance level. Ensure 'Use default compliance settings' is checked. You will probably be asked to rebuild after changing this.

In the steps that follow, we'll be creating an application plugin, and using it to create the database schema shown above. However, we must be sure that the plugin code we write will be picked up by the MiniMiser plugin loader, and that the database creation code will work as we expect.

Enter TDD.

Test-Driven Development / Test-Driven Design - Briefly

Test-driven development is a method for ensuring your code works as expected. Another way of looking at it is that it's a design activity, and that you use the writing of tests to drive the creation of your code, and in doing so, you ensure that your code is not too tightly coupled to the rest of the system, and that each aspect you develop is testable in isolation. There are several excellent articles on TDD on the IBM DeveloperWorks site, and a couple of good books available, including TDD in Action from Manning Press.

The TDD cycle works as follows:

- Red. You write a test that fails. This may drive the creation of new code (classes, methods), but you don't fill them in yet. Write a test for something you want to verify, run it, and see it fail. If you're using JUnit in Eclipse, you'll see the infamous 'red bar' when your test fails.
- Green. You write the simplest code that causes the test to pass. Running the test gives you the green bar.
- Refactor. Remove any duplication, get CheckStyle happy about your code, refactor if any metrics show signs of complexity.
- Iterate...

Working in this way gives you rapid feedback that the code you're writing is good. Ideally, use it to drive the development of all code. The MiniMiser framework was mostly written using TDD – the GUI parts were tested visually.

Another useful way of ensuring code quality is to use a coverage analysis tool. This watches the code as it's being executed by your tests, and shows you which parts are being exercised, and more importantly, which areas are not. I use EclEmma inside Eclipse, and Cobertura executed by Maven on a Hudson Continuous Integration server for this.

This short section cannot go into more detail on TDD, however, you should look at JUnit and EasyMock as they'll make testing much easier.

Developing the sample application

This part covers the following aspects of the framework:

- Creating plugins
- Defining your database schema
- Building the data access layer using the Data Access Object (DAO) pattern
- Creating a view and controller
- Using third-party plugins

The section concludes with how to build a releasable product.

Creating the application plugin

The first test we need to write will verify that MiniMiser is loading our plugin. We'll use some of MiniMiser's own test code to load the plugins that will be available when the framework application loads.

In the src/test/java source folder, create your own package (for this example, I choose org.devzendo.beanminder.plugin), and under there create a class called TestPluginLoading:

```
@Test
public void ourPluginIsLoaded() throws PluginException {
    final PluginHelper pluginHelper = new PluginHelper(false);
    pluginHelper.loadStandardPlugins();
    Assert.assertTrue(
            "Our plugin is not loaded",
            pluginHelper.getApplicationPlugin() instanceof BeanMinderApplicationPlugin);
}
```

In the src/main/java source folder, create your own package (for this example, I choose org.devzendo.beanminder.plugin), and under there create an implementation of the interface org.devzendo.minimiser.pluginmanager.ApplicationPlugin. To make things slightly easier, subclass org.devzendo.minimiser.pluginmanager.AbstractPlugin also. For this example, I'll use org.devzendo.beanminder.plugin.BeanMinderApplicationPlugin as this class.

Fill in the methods getName() and getVersion() to return suitable strings. For now, everything else can be empty or return null.

If you run the above test, it should fail (Red) with a PluginException stating that no application plugin has been defined. This is correct, as you have not yet provided a plugin descriptor for the above plugin. If you run the framework with no plugin descriptor, or with no application plugin defined (as detailed soon) then you will see an problem reporter dialog informing you of the missing plugin, and then the framework will terminate.

Creating the plugin descriptor

Under the src/main/resources source folder, create a new folder hierarchy called META-INF/minimiser, and in there, create a file plugins.properties.

Add the reference to your application plugin class into that file:

```
myApp=org.devzendo.beanminder.plugin.BeanMinderApplicationPlugin
```

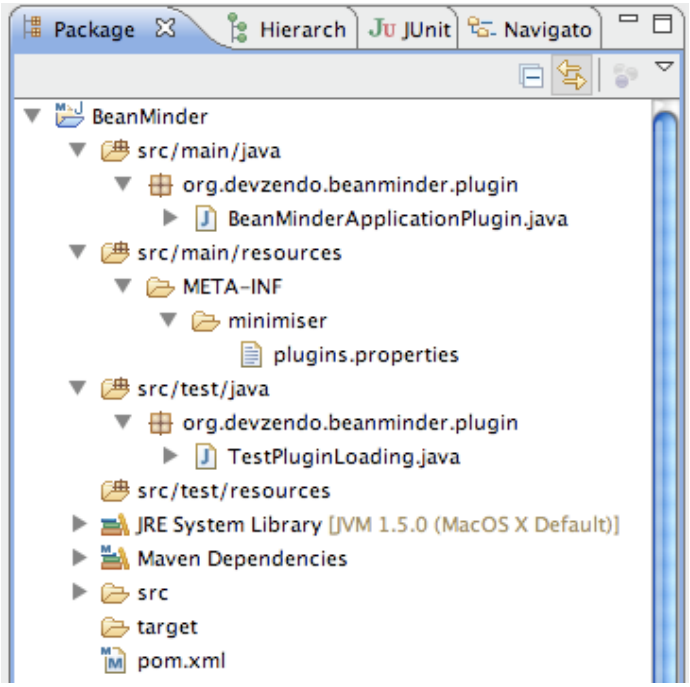
It does not matter what is on the left-hand side of the = in the plugin descriptor file, although all lines in the file must have unique left hand values.

In the plugin descriptor file, you will reference all the plugins that you provide in your project. There can be only one plugin descriptor per project, so if you are building up your application from many plugins, it might be best to create each plugin in a separate project. The framework will scan the classpath upon starting to find all META-INF/minimiser/plugins.properties files, and from there, load all classes.

Running the above test again once the plugin descriptor is in place should yield success (Green) – the framework has correctly detected your plugin. There's nothing to Refactor yet.

Overview of the project directory structure

When done, your package hierarchy should look something like this:



Running your plugin with MiniMiser

With your new project selected, create a new Java Application Run Configuration:

Right-click on the project, Run As >, Run Configurations...

In the left-hand list of Run Configurations, select Java Application, then press the 'New' button (looks like a blank page of paper).

Change the Name to BeanMinderApp.

In the Main tab:

The Project should be BeanMinder

The main class set to org.devzendo.minimiser.MiniMiser.

Optional:

In the Arguments tab, it may be useful to set the program arguments to -debug -threads -classes -times (see later section on Logging on page 51)

For Mac OS X, you will need initialise the java.library.path to include the location of the Quaqua look-and-feel native libraries, and set the name of the application as shown in its main menu and on the dock. These steps are done with the following VM arguments (if using the CrossPlatformLauncherPlugin - see later):

```
-Djava.library.path=${project_loc:BeanMinder}/target/macosx/BeanMinder.app/Contents/Resources/Java/lib
-Xdock:name=BeanMinder
```

If not using the CrossPlatformLauncherPlugin, the Quaqua native libraries will need to be unpacked from their respository location ($HOME/.m2/repository/ch/randelshofer/libquaqua/6.5/libquaqua-6.5.zip) into some temporary directory, then set java.library.path to this temporary directory.

In the Common tab, you can choose to save this Run Configuration (and possibly check it into version control) – choose Save As / Shared File, and browse to the BeanMinder project (the shared file path should then read /BeanMinder). This will create a file in the root of your project called BeanMinderApp.launch. It's also useful to check the 'Display in favourites menu / Run' checkbox to allow rapid launching.

Then Apply this configuration.

Upon running this, you should see the main application window, from which you can create databases.

Congratulations! You have started your MiniMiser application!

Now we need to define the database schema that will be used to hold the data for the BeanMinder project.

Creating the database schema

When a user creates a new database – i.e. a set of H2 database files within a directory - with BeanMinder, the framework will create some objects of its own (the Versions table, the Sequence sequence) and populate them with initial information.

We need to tell the framework about the database schema we need for BeanMinder, and hook into the database initialisation process to create these tables. Look back at the UML diagram to see what we're trying to create.

First, a test to ensure that the relevant tables are being created. We'll use some of the framework's testing code to help with this. This code (the PersistencePluginHelper) needs to know where to create its temporary databases. To do this, it reads a file called testconfig.properties, which must be located in the current directory when the test runs – by default, this will be the root directory of your project, alongside the pom.xml. The file gives the path to a directory that must exist and be empty prior to the test running. This directory will be cleared after the test runs. In my sample project, my testconfig.properties looks like:

```
database.directory=/Users/Shared/Temp/beanminder
```

It must read database.directory=... some path ...

Then, create a test in src/test/java, with a package of org.devzendo.beanminder.persistence, containing:

```
public final class BeanMinderDatabaseCreatedCorrectly {
    private static final String DBNAME = "sadatabase";
    private static final String DBPASSWORD = "";
    private PersistencePluginHelper mPersistencePluginHelper;

    @Before
    public void getPrerequisites() throws PluginException {
```

```
        final PluginHelper pluginHelper = PluginHelperFactory.createMiniMiserPluginHelper();
        mPersistencePluginHelper = new PersistencePluginHelper(false, pluginHelper);
        mPersistencePluginHelper.validateTestDatabaseDirectory();
        pluginHelper.loadStandardPlugins();
    }

    @After
    public void tidyDatabases() {
        mPersistencePluginHelper.tidyTestDatabasesDirectory();
    }

    @Test
    public void areTablesCreated() {
        final InstanceSet<DAOFactory> database = mPersistencePluginHelper.createDatabase(DBNAME, DBPASSWORD);
        final MiniMiserDAOFactory miniMiserDAOFactory =
            database.getInstanceOf(MiniMiserDAOFactory.class);
        try {
            final SimpleJdbcTemplate jdbcTemplate = miniMiserDAOFactory.
                    getSQLAccess().getSimpleJdbcTemplate();

            final int accounts = jdbcTemplate.queryForInt("select count(*) from Accounts");
            Assert.assertTrue(accounts == 0);

            final int transactions = jdbcTemplate.queryForInt("select count(*) from Transactions");
            Assert.assertTrue(transactions == 0);

        } finally {
            miniMiserDAOFactory.close();
        }
    }
}
```

Some explanation:

The @Before method sets up plugin and persistence helpers to create a database.

The @After method will delete up the database files after they have been created.

The @Test itself:

Creates the database and returns an InstanceSet<DAOFactory> - this is a set of objects that can be used to operate on different aspects of the database. Each plugin in your configuration can an instance of DAOFactory; each DAOFactory provides access to that plugin's tables. Later, we'll create a BeanMinderDAOFactory from which the rest of the application will access the Accounts and Transactions tables.

The MiniMiserDAOFactory is added to this set of DAOFactory objects by the framework itself. It can be used to obtain the VersionsDAO and SequenceDAO, which allow access to the Versions table and Sequence sequence respectively. It can also be used to obtain the SqlAccess object, from where lower-level SQL operations can be carried out, in this case, using a Spring JDBC Template.

The existence of the BeanMinder tables is verified simplistically by ensuring that a COUNT(*) on them returns no rows.

If you run this test, it'll fail with a BadSqlGrammarException since the tables do not exist. (Red).

Let's make this green by creating the tables. Your ApplicationPlugin by itself provides no facility for creating tables, but by implementing other interfaces provided by the framework, it can be involved in:

- Adding panels to the 'New Database Wizard', where custom information can be entered.

- Creating its own tables after the 'New Database Wizard' has been completed, given the data entered in that wizard.
- Providing a DAOFactory upon database creation/opening to allow access to the tables created above.
- And more...

All these operations are fairly disconnected; they represent different aspects of your application: GUI, SQL, Object-Relational access, etc., so they are implemented by different objects in your application. This ensures a reasonable separation of concerns, and can aid testability. (This is discussed further in this guide, with a diagram of the classes and interfaces involved - see page 37).

To add one of these operations to your ApplicationPlugin, you implement an appropriate interface under org.devzendo.minimiser.plugin.facades. In this case, NewDatabaseCreation. Your ApplicationPlugin must then implement the getNewDatabaseCreationFacade() method, which returns a NewDatabaseCreationFacade object.

Create an implementation of NewDatabaseCreationFacade as org.devzendo.beanminder.plugin.facade.newdatabasecreation.BeanMinderNewDatabaseCreationFacade, as shown:

```
public final class BeanMinderNewDatabaseCreationFacade
    implements NewDatabaseCreationFacade {
    private static final String[] CREATION_DDL_STRINGS =
        new String[] {
        "CREATE TABLE Accounts("
                + "id INT PRIMARY KEY,"
                + "name VARCHAR(40) NOT NULL,"
                + "with VARCHAR(40),"
                + "accountCode VARCHAR(40) NOT NULL,"
                + "initialBalance INT,"
                + "currentBalance INT"
                + ")",
        "CREATE TABLE Transactions("
                + "id INT PRIMARY KEY,"
                + "accountId INT NOT NULL,"
                + "FOREIGN KEY (accountId) REFERENCES Accounts (id) ON DELETE CASCADE,"
                + "index INT NOT NULL,"
                + "amount INT NOT NULL,"
                + "isCredit BOOLEAN,"
                + "isReconciled BOOLEAN,"
                + "transactionDate DATE,"
                + "accountBalance INT"
                + ")",
    };

    public void createDatabase(
            final DataSource dataSource,
            final SimpleJdbcTemplate jdbcTemplate,
            final Observer<PersistenceObservableEvent> observer,
            final Map<String, Object> pluginProperties) {
        for (int i = 0; i < CREATION_DDL_STRINGS.length; i++) {
            observer.eventOccurred(new PersistenceObservableEvent(
                "Creating BeanMinder database..."));
            jdbcTemplate.getJdbcOperations().
                execute(CREATION_DDL_STRINGS[i]);
        }
    }

    public int getNumberOfDatabaseCreationSteps(
            final Map<String, Object> pluginProperties) {
        return CREATION_DDL_STRINGS.length;
    }
```

```
    public void populateDatabase(
            final SimpleJdbcTemplate jdbcTemplate,
            final SingleConnectionDataSource dataSource,
            final Observer<PersistenceObservableEvent> observer,
            final Map<String, Object> pluginProperties) {
        // TODO Nothing here yet
    }
}
```

Some points to note about this code before continuing with connecting it up.

It creates the necessary database tables and indices in the createDatabase() method, and fills any of these tables with initial data in the populateDatabase() method (nothing is populated in this example yet).

In a real application, it would be better to store the DDL used to create the database as a resource, one statement per line, executing each line sequentially, to provide some separation between the Java and SQL.

User feedback during creation is provided via the getNumberOfDatabaseCreationSteps() method, and the calls to observer.eventOccurred(...) in createDatabase(). There is a progress bar shown when the user creates a database, which gradually increases towards completion as the creation progresses. The getNumberOfDatabaseCreationSteps() method is called prior to any database creation/population, and should return the total number of times you intend to call the observer.eventOccurred(...) method – the progress bar creation code needs to know how many steps there are in total. Each step in your database creation and population should make a call to observer.eventOccurred(...) to advance the progress bar and display an appropriate message.

Let's connect it in. Create an instance of this in your ApplicationPlugin, like this:

```
public class BeanMinderApplicationPlugin extends AbstractPlugin
    implements ApplicationPlugin, NewDatabaseCreation {

    private final BeanMinderNewDatabaseCreationFacade mNewDatabaseCreationFacade;

    public BeanMinderApplicationPlugin() {
        mNewDatabaseCreationFacade =
            new BeanMinderNewDatabaseCreationFacade();
    }

    public String getName() {
        return "BeanMinder";
    }

    public String getSchemaVersion() {
        return "0.1";
    }

    public String getVersion() {
        return "0.1";
    }

    public NewDatabaseCreationFacade getNewDatabaseCreationFacade() {
        return mNewDatabaseCreationFacade;
    }

    public String getAboutDetailsResourcePath() {
        // TODO Auto-generated method stub
        return null;
    }

    public String getChangeLogResourcePath() {
        // TODO Auto-generated method stub
        return null;
    }
```

```
    public String getDevelopersContactDetails() {
        // TODO Auto-generated method stub
        return null;
    }

    public String getFullLicenceDetailsResourcePath() {
        // TODO Auto-generated method stub
        return null;
    }

    public String getIntroPanelBackgroundGraphicResourcePath() {
        // TODO Auto-generated method stub
        return null;
    }

    public String getShortLicenseDetails() {
        // TODO Auto-generated method stub
        return null;
    }

    public String getUpdateSiteBaseURL() {
        // TODO Auto-generated method stub
        return null;
    }

    public List<String> getApplicationContextResourcePaths() {
        // TODO Auto-generated method stub
        return null;
    }

    public void shutdown() {
        // TODO Auto-generated method stub
    }
}
```

Also note that I've implemented getName(), getVersion() and getSchemaVersion(), and return the instance of NewDatabaseCreationFacade.

The test we wrote previously should now succeed. You may want to enhance it to determine the correct structure of your database.

Using the SpringLoader

In the previous section, the BeanMinderNewDatabaseCreationFacade object was instantiated directly in the constructor of the ApplicationPlugin. This can be changed to load a Singleton instance of it from a Spring application context, using the SpringLoader.

When your ApplicationPlugin is initialised, it will be given the opportunity to define its own application context files, and store a reference to the SpringLoader, which is a thin wrapper around the Spring ClasspathXmlApplicationContext object.

Create the application context file under src/main/resources/org/devzendo/beanminder, called BeanMinder.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd"
    default-lazy-init="true" >

    <bean id="newDatabaseCreationFacade"
          class="org.devzendo.beanminder.plugin.facade.newdatabasecreation. '
```

```
BeanMinderNewDatabaseCreationFacade" />
</beans>
```

Then change your ApplicationPlugin to refer to this application context, and delegate the loading of the NewDatabaseCreationFacade object to the SpringLoader at the point of need:

```
public class BeanMinderApplicationPlugin extends AbstractPlugin
    implements ApplicationPlugin, NewDatabaseCreation {

    public BeanMinderApplicationPlugin() {
    }

    public NewDatabaseCreationFacade getNewDatabaseCreationFacade() {
        return getSpringLoader().getBean("newDatabaseCreationFacade",
            NewDatabaseCreationFacade.class);
    }

    public List<String> getApplicationContextResourcePaths() {
        return Arrays.asList(new String[] {
                "org/devzendo/beanminder/BeanMinder.xml"
        });
     }
    // the rest remains the same...
```

Creating the Data Access Object layer

The domain layer and the DAO pattern

Although your application's data is stored in a relational database, this does not mean that you scatter SQL access code throughout your code. It is better to separate your application into layers, with a *data access layer* insulating the SQL and underlying relational tables from the rest of the the application.

This approach allows the underlying persistence technology to be changed if needed.

Typically, for simple databases, JDBC access using Spring JDBC is sufficient. Spring JDBC makes writing database access code far easier than with raw JDBC. You will need to implement code that performs CRUD (Create, Read, Update, Delete) operations on the database, transforming the data returned by JDBC into objects to be used by the rest of your application. This is the approach taken in this tutorial, with a fairly extensive illustration of the work involved.

For more complex databases, it may be necessary to use an Object-Relational Mapping framework such as Hibernate (which Spring also makes easier).

The interfaces and data transfer objects you develop as part of the data access layer are not specific to either of these approaches; they insulate you from the actual implementation.

The choice of using Spring JDBC or Spring Hibernate depends on the complexity of your database schema (greater complexity suggests using ORM), whether you intend to issue many batch operations (JDBC) ...

Your implementation of the data access layer works with the JdbcTemplate and DataSource/Connection to implement to the DAO objects/interfaces.

DAO interfaces typically contain:

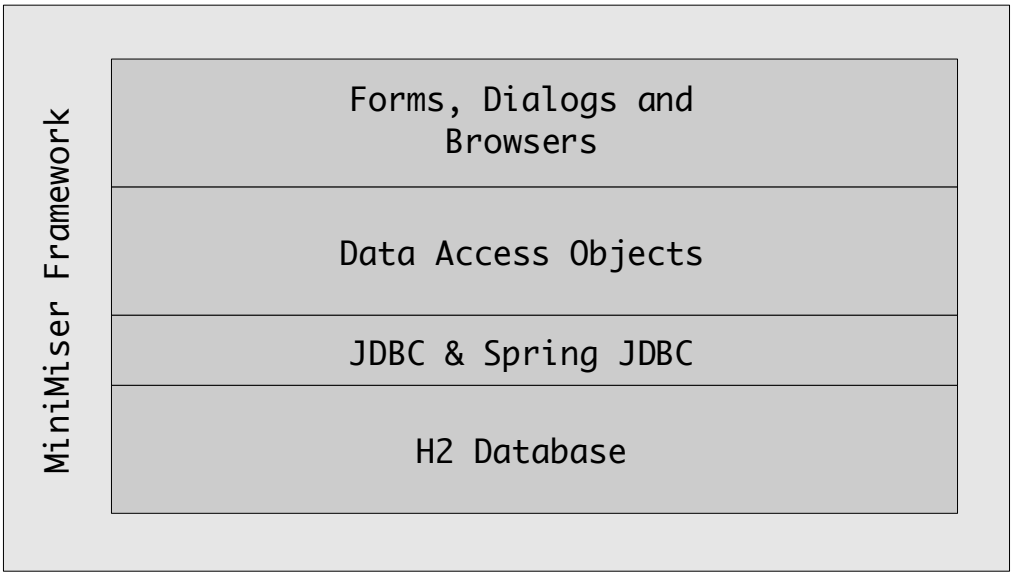
- Finder methods; allow finding of objects by criteria other than by their primary key
- Object graph / aggregate methods

- CRUD – Create, Read (by primary key), Update, Delete methods

They do not typically contain transaction-related code; this is usually implemented at a higher level.

As they are interfaces to which you provide a concrete implementation, they should also be easy to implement using mock objects, to allow other layers of the application to be tested without needing a real database. Since H2 is embedded, it is also easy to test against a real database – in some cases, necessary.

The layering looks like this:



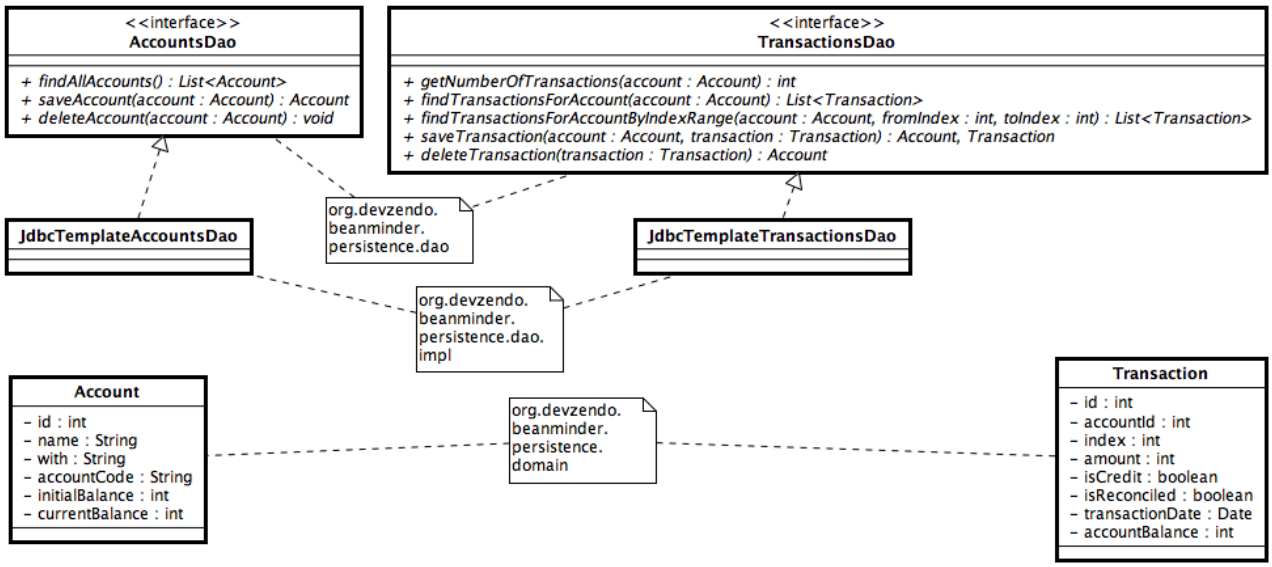
The structure of the DAO layer can be either fine-grained (a thin abstraction layer atop the tables and CRUD operations on them), or coarse-grained (persisting entire object graphs; a style more suited to the use of ORM). Each DAO interface is responsible for one domain object; if a domain object has an independent lifecycle, it should have its own DAO interface.

The example here will use the former, fine-grained style. We'll define DAO interfaces that describe the operations for each table in our sample application, and build domain objects (Data Transfer or Value Object pattern) that map onto rows in the tables, and that are the objects that the DAO interfaces work with. We'll build implementations of the DAO interfaces using Spring JDBC templates.

It's important to note that at this stage in the development of the DAO layer, it is independent of the MiniMiser framework: the only interfaces and objects you use are from the Spring framework – notably, JdbcTemplate and the Spring exception hierarchy.

The sample application DAO layer

Looking back at the description of the sample application, and the functionality required, the following objects and methods are needed to comprise this independent portion of the DAO layer. Also note the package hierarchy and fine-grained nature of the operations:



Noteworthy points here are that:

- inside the JdbcTemplateXXXDao, whenever an object is saved, it is either inserted on first creation, or updated subsequently; the code that uses the DAO layer does not need to know which operation occurs, this is a necessary distinction in the SQL used to effect these operations, but is an irrelevance to client code.
- to save a new object, (Account or Transaction), an instance of the Account or Transaction bean is created without the id fields set, initialised appropriately, then passed to one of the save methods on the appropriate Dao object. The save method then returns the object, with the id fields filled in.
- the id fields are primary and foreign keys - id in Account and Transaction; accountId in Transaction – they're used internally by the AccountsDao and TransactionsDao, but have no meaning; they are surrogate keys.
- whenever a Transaction is added to an Account, the Account's current balance is adjusted by the Transaction's amount and persisted. Both the Transaction (with id fields filled in), and the adjusted Account are returned. This enables client code to update its view of the Account balance. The returned Transaction also contains the current account balance so that the ongoing balance of the Account can be displayed against each Transaction.
- Transactions are ordered by their transaction index, which starts at zero and increments. The TransactionsDao uses the transaction index to provide array-like semantics for the transactions in an account. The number of transactions in an account can be obtained; An account's transactions can be returned en masse, or by a range of indices; They can be deleted by index, with the account and all subsequent transactions being updated with the

new balances; New transactions can be appended to the account, or inserted before any other Transaction (with the balances of the account and other transactions being updated accordingly).

- although the Dao objects try to insulate the client code from the underlying relational database, this is not perfect – this implementation does not offer true transparent persistence, as would, for example Hibernate. This can be observed in the Account and Transaction objects. These are javabeans, a.k.a. POJOs – Plain Old Java Objects. In a transparent persistence system, to add a transaction to an account, one would have an addTransaction(Transaction transaction) method on the Account object. Calling this would transparently insert the row in the TRANSACTIONS table, and update the account in the ACCOUNTS table. Here, Account and Transactions are purely Data Transfer Objects or Value Objects – they do not contain any persistence code, nor do they have any "connection back" to the Dao objects that manage them. All operations involving them are mediated by the Dao objects. Accounts are managed by the AccountsDao; Transactions are managed by the TransactionsDao – e.g. to add a Transaction to an Account, the Account and Transaction are passed to the TransactionsDao. This is counter to how you might use a transparent persistence system, and what you might consider "correct" Object-Oriented design.

Dependency Injection vs Multiple Databases Open At Once

If you have used Spring JDBC in traditional enterprise development, using a centralised, single instance of a pre-deployed database, for example MySQL or SQL Server, you need to be aware of a difference in the way you use Spring JDBC with that configuration, and how it is used in the MiniMiser framework.

In traditional, enterprise Java development with Spring and Spring JDBC Templates, Spring will handle many of the details of getting a Connection to the database, after you have configured a DataSource. In this setting, your DAO layer objects are declared in the Application Context, and when they are instantiated by Spring, they will have the DataSource automatically injected, from where a JdbcTemplate can be constructed and stored in an instance variable for later use. Spring manages the single instances of the objects in the DAO layer, and the database connectivity objects.

In the MiniMiser framework, there can be multiple databases open at once, with the user switching between them at will. There is no single DataSource or JdbcTemplate – there is a separate instance of these per open database. It is the MiniMiser framework's responsibility to track these in its open database list.

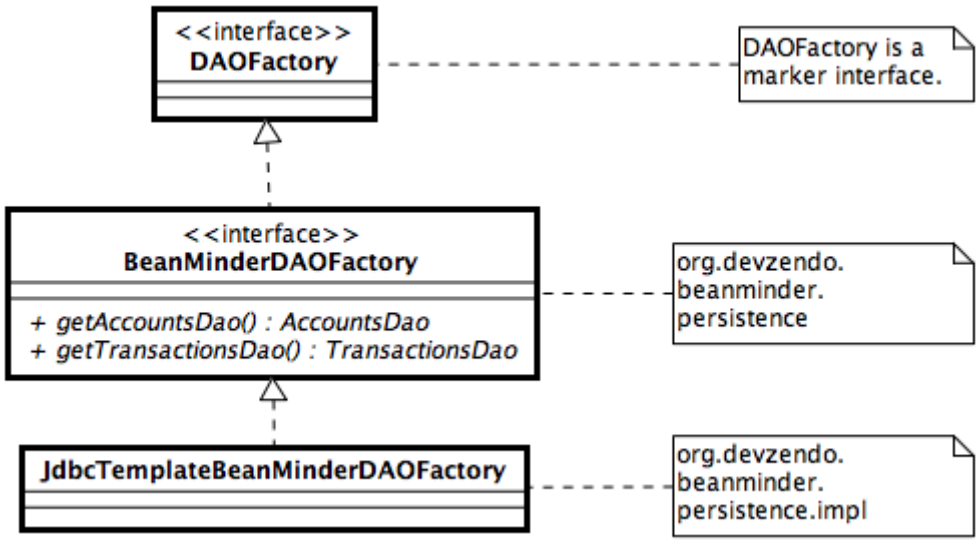
During database creation the framework will inject the DataSource and JdbcTemplate into the NewDatabaseCreationFacade via the createDatabase(...) and populateDatabase(...) methods. After creation, and during database opening, the DataSource and JdbcTemplate objects are passed to the createDAOFactory(...) method of the DatabaseOpeningFacade. This returns a DAOFactory that is stored in the open database list, alongside the other plugins' DAOFactory instances, and other internal objects that comprise an open database (mostly GUI components for each database).

This is why the DAO layer is not defined in an application context in MiniMiser, and why it is injected by MiniMiser, not Spring.

Making use of the DAO layer with the framework

As mentioned above, at this point, the DAO layer is not coupled to the MiniMiser framework. However, in order to make use of it, you must define a subinterface of DAOFactory, and provide an implementation of it. This couples your DAO layer to the MiniMiser framework.

The DAOFactory instantiates the persistence-technology-specific DAO classes. In the example above, you create a BeanMinderDAOFactory interface, and an implementation of it: JdbcTemplateBeanMinderDAOFactory. This implementation creates instances of JdbcTemplateAccountsDao and JdbcTemplateTransactionsDao.



Driving the development of the DAO layer with tests

It's time to write tests that allow the development of the DAO layer. The first test ensures that we can create an account, and that it has no transactions initially. With this in place, we can fill in the classes and interfaces shown above, and the JdbcTemplateXXXDao and JdbcTemplateBeanMinderDAOFactory objects.

The first task is to extract the database creation test code from the earlier test into a superclass, and make use of it in our new test class, TestAccountsDao. I won't show this step, it should be apparent from the usage in the new test class, nor will I show the interfaces and POJOs described earlier – they're exactly as shown in the class diagram. With empty 'do-nothing' implementations of these (e.g. just implement the interfaces of AccountsDao in JdbcTemplateAccountsDao, returning null from all methods), the test will fail with a NullPointerException, as the DatabaseOpeningFacade has not been implemented and attached to the BeanMinderApplicationPlugin.

```
public final class TestAccountsDao extends BeanMinderDatabaseTest {
    @Test
    public void createEmptyAccount() {
        final InstanceSet<DAOFactory> database =
            getPersistencePluginHelper().createDatabase(DBNAME, DBPASSWORD);
        final BeanMinderDAOFactory simpleAccountsDaoFactory =
            database.getInstanceOf(BeanMinderDAOFactory.class);
        final AccountsDao accountsDao = simpleAccountsDaoFactory.getAccountsDao();

        final Account newAccount = new Account("Test account", "123456", "Imaginary Bank of London", 5600);
```

```
        final Account savedAccount = accountsDao.saveAccount(newAccount);

        Assert.assertTrue(savedAccount.getId() > 0);
        Assert.assertEquals(newAccount.getAccountCode(), savedAccount.getAccountCode());
        Assert.assertEquals(newAccount.getName(), savedAccount.getName());
        Assert.assertEquals(newAccount.getWith(), savedAccount.getWith());
        Assert.assertEquals(newAccount.getBalance(), savedAccount.getBalance());

        final TransactionsDao transactionsDao = simpleAccountsDaoFactory.getTransactionsDao();
        final List<Transaction> transactions = transactionsDao.findAllTransactionsForAccount(savedAccount);
        Assert.assertEquals(0, transactions.size());
    }
}
```

Working through this test's requirements now, let's implement the DatabaseOpeningFacade:

```
package org.devzendo.beanminder.plugin.facade.databaseopening;

public final class BeanMinderDatabaseOpeningFacade implements
        DatabaseOpeningFacade {
    public InstancePair<DAOFactory> createDAOFactory(
            final SimpleJdbcTemplate jdbcTemplate,
            final SingleConnectionDataSource dataSource) {
        final BeanMinderDAOFactory daoFactory =
            new JdbcTemplateBeanMinderDAOFactory(jdbcTemplate);
        return new InstancePair<DAOFactory>(BeanMinderDAOFactory.class, daoFactory);
    }
}
```

Wire this into the application context:

```
        <bean id="databaseOpeningFacade"
        class="org.devzendo.beanminder.plugin.facade.databaseopening.BeanMinderDatabaseOpeningFacade" />
```

Add the new Facade to the Application Plugin:

```
public class BeanMinderApplicationPlugin extends AbstractPlugin
    implements ApplicationPlugin, NewDatabaseCreation, DatabaseOpening {
    ...
    public DatabaseOpeningFacade getDatabaseOpeningFacade() {
        return getSpringLoader().getBean("databaseOpeningFacade",
            DatabaseOpeningFacade.class);
    }
    ...
```

And add the JdbcTemplateBeanMinderDAOFactory:

```
package org.devzendo.beanminder.persistence.impl;
public final class JdbcTemplateBeanMinderDAOFactory implements BeanMinderDAOFactory {

    public JdbcTemplateBeanMinderDAOFactory(final SimpleJdbcTemplate jdbcTemplate) {
    }

    public AccountsDao getAccountsDao() {
        return null;
    }

    public TransactionsDao getTransactionsDao() {
        return null;
    }
}
```

The test should now get a little further – now to implement the AccountsDao and TransactionsDao, and instantiate them in the JdbcTemplateBeanMinderDAOFactory.

```
public final class JdbcTemplateBeanMinderDAOFactory implements
        BeanMinderDAOFactory {

    private final JdbcTemplateAccountsDao mAccountsDao;
```

```
    private final JdbcTemplateTransactionsDao mTransactionsDao;

    public JdbcTemplateBeanMinderDAOFactory(final SimpleJdbcTemplate jdbcTemplate) {
        mAccountsDao = new JdbcTemplateAccountsDao(jdbcTemplate);
        mTransactionsDao = new JdbcTemplateTransactionsDao(jdbcTemplate);
    }

    public AccountsDao getAccountsDao() {
        return mAccountsDao;
    }

    public TransactionsDao getTransactionsDao() {
        return mTransactionsDao;
    }
}
```

All painless so far. Now the AccountsDao implementation, just enough to be able to save a new account - this will change as we add more tests:

```
package org.devzendo.beanminder.persistence.dao.impl;
public final class JdbcTemplateAccountsDao implements AccountsDao {
    private final SimpleJdbcTemplate mJdbcTemplate;
    public JdbcTemplateAccountsDao(final SimpleJdbcTemplate jdbcTemplate) {
        mJdbcTemplate = jdbcTemplate;
    }
    ...
    public Account saveAccount(final Account account) {
        if (account.getId() != -1) {
            return updateAccount(account);
        } else {
            return insertAccount(account);
        }
    }

    private Account insertAccount(final Account account) {
        final KeyHolder keyHolder = new GeneratedKeyHolder();
        mJdbcTemplate.getJdbcOperations().update(new PreparedStatementCreator() {
            public PreparedStatement createPreparedStatement(final Connection conn)
                    throws SQLException {
                final String sql = "INSERT INTO Accounts (name, with, accountCode, initialBalance, currentBalance)
VALUES (?, ?, ?, ?, ?)";
                final PreparedStatement ps = conn.prepareStatement(sql, new String[] {"id"});
                ps.setString(1, account.getName());
                ps.setString(2, account.getWith());
                ps.setString(3, account.getAccountCode());
                ps.setInt(4, account.getInitialBalance());
                ps.setInt(5, account.getCurrentBalance());
                return ps;
            }
        }, keyHolder);
        final int key = keyHolder.getKey().intValue();
        return new Account(
            key, account.getName(), account.getAccountCode(), account.getWith(), account.getBalance());
    }

    private Account updateAccount(final Account account) {
        // TODO Auto-generated method stub
        return null;
    }
}
```

And the TransactionsDao implementation, again, just enough to pass the test – so, finding transactions for an account:

```
package org.devzendo.beanminder.persistence.dao.impl;
public final class JdbcTemplateTransactionsDao implements TransactionsDao {
    private final SimpleJdbcTemplate mJdbcTemplate;
    public JdbcTemplateTransactionsDao(final SimpleJdbcTemplate jdbcTemplate) {
        mJdbcTemplate = jdbcTemplate;
```

```
    }

    public List<Transaction> findAllTransactionsForAccount(final Account account) {
        final String sql = "SELECT id, accountId, index, amount, isCredit, isReconciled, transactionDate, "
            + "accountBalance FROM Transactions WHERE accountId = ?";
        final ParameterizedRowMapper<Transaction> mapper = new ParameterizedRowMapper<Transaction>() {
            public Transaction mapRow(final ResultSet rs, final int rowNum) throws SQLException {
                return new Transaction(
                    rs.getInt("id"),
                    rs.getInt("accountId"),
                    rs.getInt("index"),
                    rs.getInt("amount"),
                    rs.getBoolean("isCredit"),
                    rs.getBoolean("isReconciled"),
                    rs.getDate("transactionDate"),
                    rs.getInt("accountBalance"));
            }
        };
        return mJdbcTemplate.query(sql, mapper, account.getId());
    }
    ...
}
```

The test now passes, but there's much more work to do in the DAO layer.

The TransactionsDao implementation is quite involved, having to implement the array-like semantics required, updating the Account and Transactions as necessary, across insertions, updates, and deletions.

Rather than continuing to include the code and tests for the complete DAO layer in a walk-through style in this guide, please consult the sample application source code, available from the DevZendo.org website. The majority of it is standard DAO-style JDBC code, and is not specific to the MiniMiser framework.

With the DAO layer in place, it's time to work on the user interface.

Creating the User Interface

(some words on how MiniMiser builds much of the UI for you)

Customising the menu

(add the Accounts/New... menu, menu item, and the new account dialog)

Creating View menu items

(add each account's name to the View menu, wired to the account Tab)

Creating Tabs

(add the account Tab)

Part III – MiniMiser Framework Reference

Programming languages

At the moment, to develop your plugins or application, only Java is supported, although it is possible to use the Spring framework's support for other dynamic languages. The classes declared in the META-INF/minimiser/plugins.properties file must be Java classes.

It is hoped to be able to develop plugins completely in JRuby, in a later release. Some initial work on this has been completed; the framework needs to use Spring 3.0 and a later version of JRuby for this to be viable.

During this discussion of the plugin architecture, you should consult the JavaDoc for org.devzendo.minimiser.plugin.Plugin and org.devzendo.minimiser.plugin.ApplicationPlugin.

Plugin architecture

An application using the framework must provide a single Application Plugin – this must implement org.devzendo.minimiser.plugin.ApplicationPlugin, and can make use of AbstractPlugin from the package to take care of some of the boilerplate code.

You may also write several 'normal', non-application plugins by implementing org.devzendo.minimiser.plugin.Plugin, again subclassing AbstractPlugin as necessary.

Scanning for plugins.properties files

Plugin loading is done dynamically at framework startup. The framework searches for any META-INF/minimiser/plugins.properties files available on the classpath, and from these, determines the classes that need to be instantiated. These properties files may be present in any .jar file on the classpath.

The format of the file is

```
xxxx=my.plugin.package.Plugin
yyyy=my.other.plugin.package.PluginForWhatever
zzzz=my.application.pluginb.MyApplication
```

Where xxxx doesn't matter, but must be unique in the file – it could be numeric, or could be the same as the name of the plugin. The rest of the line is the fully-package-qualified name of the plugin class.

Instantiating plugin classes

Each class is instantiated using its zero-arg constructor.

Once all plugins are loaded and instantiated, with no PluginExceptions being thrown[4], the plugins will be initialised.

4 PluginExceptions will result in the Minimiser framework's Problem Reporter dialog being shown.

Initialising plugin objects

After all plugin classes have been instantiated, they are initialised. Checks are made that there are actually some plugins to initialise, and that there is one and only one ApplicationPlugin.

The framework makes use of the Spring framework to manage instantiation of many of its objects, maintaining them as Singletons or creating new instances as necessary. Many of these framework objects are not available to plugins, and are internal to the framework. Some, however, may be used by plugins.

This document does not attempt to provide any documentation for Spring – there is plenty of information available available for Spring. In particular, Spring in Action by Manning Press is an excellent book on the subject.

The plugins' getApplicationContextResourcePaths() method is then called. This gives the plugin the ability to define its own Spring application context XML files. If any are required, they are added as children to the framework's own application contexts. Multiple application context XML files can be provided, giving your the ability to define your beans in files relating to specific layers: persistence, GUI, etc.

The SpringLoader object is then given to the plugin objects by calling setSpringLoader(). The SpringLoader is a thin wrapper around Spring's Application Context bean-loading mechanism. By using it, you will be able to load your beans. When this is given to your plugin, just stash it away; do not perform any bean loading immediately.

The getName(), getVersion(), getSchemaVersion() and other 'metadata' methods of your plugin will then be called to obtain its name and version, schema version (if it provides any database tables/data). If your plugin is an ApplicationPlugin, it will also be queried for any update site URL, developer contact details, license text and 'About' text for the 'About' and 'Welcome' dialogs.

The name and version information is important; for the ApplicationPlugin, the name is used throughout the application to refer to it by name, e.g. in the Help menu, Welcome, About and Problem Reporter dialogs. The name, version and schema version are also persisted in the Versions table which is provided by the framework to enable it to correctly process database migrations (schema upgrades) when new versions of the application or framework are deployed against older databases.

ApplicationPlugin Resources

The plugin's name and version information are used by various parts of the framework as described above, but there are also several other attributes and resources defined by an ApplicationPlugin that are used in the Problem Reporting, About and Welcome dialogs, the Introduction panel (shown when no databases are open), the New / Open wizard[5], and the update notification system.

These are either Strings that your ApplicationPlugin returns directly, or resource paths. Resource paths are directories under src/main/resources, which is packaged by Maven into your ApplicationPlugin's .jar file. For example, hopefully you will be shipping your application under an open source license. The full text of your license could be stored at src/main/resources/org/devzendo/beanminder/GPL3.html, with your ApplicationPlugin's getFullLicenseDetailsResourcePath() method returning “org/devzendo/beanminder/GPL3.html”.

5 The wizard graphic is not yet customisable.

For full details on the types of resources that you should define in your ApplicationPlugin, consult the ApplicationPlugin JavaDoc.

Separation of concerns: Developing your plugin with Facades

There are many parts to a complete MiniMiser application, with the Plugin and Application Plugin being the place from which all these parts are created and presented to the framework.

Examples of these parts are:

- persistence-related (e.g. creating specific tables and populating them, or defining a data access layer), or
- user interface elements (e.g. menus, or a set of forms for data entry or browsers/charts)
- or somewhere in-between (e.g. prompting the user for configuration information at the start of database creation)

Each of the parts you may want to provide are provided by your plugin.

However, these parts are not to be coupled together – all data entry forms are UI-related, and should not be in the same object as your custom table creation / population code.

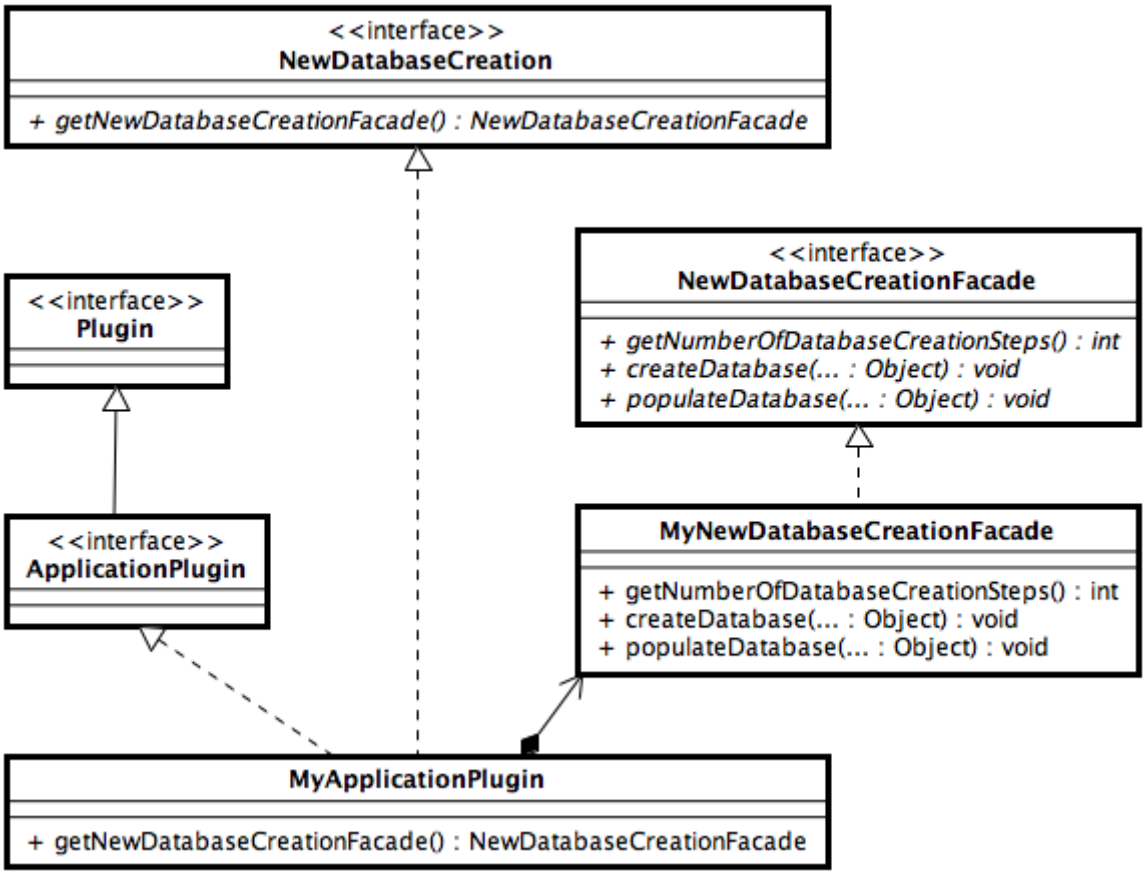
To achieve this separation of concerns, the Plugin / Application Plugin interfaces define a small core of information that is given to the framework for all plugins – name, version and licence/copyright information, as described above, with all the rest of the application being divided up into *facades*.

These are simply groupings of related functionality into an interface.

For example, if your Plugin or Application Plugin needs to create and populate its own database tables when a new database is created, it must provide an object that implements the NewDatabaseCreationFacade interface. This contains several methods that you need to provide for the framework to call when a database is created.

The linkage between your plugin and this facade is that your plugin must implement the NewDatabaseCreation interface, then provide an implementation of this interface's single method that returns your NewDatabaseCreationFacade object.

In summary, your plugin provides the entry point (implements NewDatabaseCreation) to the actual functionality (provided by an object that implements NewDatabaseCreationFacade).



Your facade objects (e.g. MyNewDatabaseCreationFacade in the diagram) can be requested many times by the framework, so it's best to store a single instance of each facade – possibly lazily-instantiating them on first request.

A better way of managing the single instance of facade implementations is to use the SpringLoader to do it, declaring them in an application context that you return from getApplicationContextResourcePaths().

Persistence

The user can have multiple databases open simultaneously, and switch between them using the Window menu. Databases can be explicitly closed from the File menu, either individually, or en masse. Any databases that are open when the application is closed are noted, and will be re-opened automatically upon application restart. Upon opening an encrypted database, the user will be prompted for the password. Such a database will not open unless the correct password is given. This password prompt will be given either upon explicit opening of an encrypted database, or upon the automatic re-opening of a database that was left closed upon previous application exit. A list of recently-opened databases is provided, for rapid access.

When a database is created, several wizard screens are presented, some from the framework; some from plugins that choose to be involved in database creation. These wizard screens allow application-specific default settings for the application to be prompted for. After the wizards, the database is created, the framework's own tables are created and populated (mostly just version

information), and any plugins' tables are created and populated – for those plugins that choose to be involved in database creation. The user is given feedback of the progress of database creation.

When databases are opened, the versions of the plugins that created the database (that are persisted in the database) are compared against the currently running plugins. If there are any differences (i.e. if there has been a software upgrade, or if opening a database created with an earlier version of the software), a migration will be performed. Each plugin – that chooses to be involved in database migration - is asked to upgrade its tables to be usable with the current version of the code. The current versions of the plugins are then recorded, to facilitate any subsequent migrations.

Access mechanisms

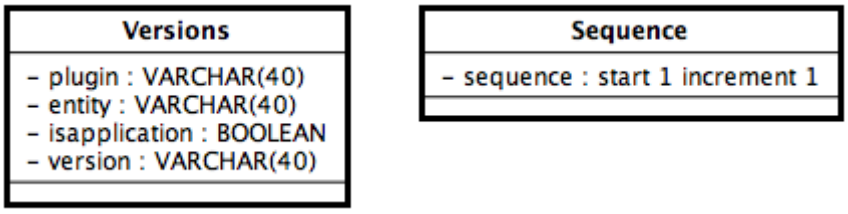
The framework uses Spring JDBC to manage database connections. The framework itself maintains a small amount of information in each database, pertaining to the versioning of plugin code and plugin database schemas.

Although the framework's raison d'être is to aid the creation of desktop GUIs around embedded relational databases, sometimes, other persistent data structures may be more appropriate. The framework allows plugin-definable files to be attached to the database files – they're stored in the same directory as the database. These files can avail themselves of the encryption settings for the database.[6]

Database schema for version control

The database schema used by the framework itself is minimal – it exists to provide a reliable incrementing sequence for use in primary keys, and to record the code and schema versions of all of the plugins in your application – irrespective of whether they partake in database creation.

6 This is on the wish list – it's just databases at the moment.



During database creation, the tables above are created. The Sequence table is initialised to produce the value 1, and the Versions table populated with the code and schema versions for each plugin, and the framework itself. The single application plugin is denoted by a *true* in the *isApplication* field – all other plugins are denoted as *false* here.

The following sections describe the creation, migration and access facilities described above in detail.

Database creation facade

Also see Database creation wizard pages facade, on page 43 in the section on Presentation.

As described above, a new MiniMiser database contains very little apart from versioning information. In order to create your own objects in a new database, your plugin must implement the NewDatabaseCreation implementation, and provides an instance of NewDatabaseCreationFacade when getNewDatabaseCreationFacade() is called.

A worked example of the mechanism involved can be found in the sample application, on page 22.

Essentially, by implementing this interface, your plugin can:

- specify the number of database tables that will be created (in order to provide a progress bar, representing the amount of work involved in database creation). In order to provide an idea of the number of objects to be created, the facade is given the map of settings entered from the New Database Wizard Pages.
- create any tables/sequences/views/triggers needed by the plugin. In order to do this, the facade is given a low-level connection to the database (a SimpleJdbcTemplate and DataSource). The database has already been populated with the framework's versioning information as described earlier. Again, the facade is given the map of settings from the New Database Wizard Pages.
- populate these tables with initial data. Low-level connections to the database are provided (SimpleJdbcTemplate and DataSource), as are the settings from the New Database Wizard Pages.

This facade is only used for issuing database operations (e.g. DDL / DML statements) for creating new databases. After the database is created, if the plugin also implements the DatabaseOpening facade, then this is used to create the objects of the DAO layer needed to access the database at a high level. See Database opening, on page 41 for further details.

Database opening facade

A database can be opened by a variety of mechanisms:

- it was open the last time the application was run, the File/Exit menu was used without using File/Close first (which is perfectly OK), and the framework is automatically re-opening it for you upon startup
- you use the File/Open... menu option
- you use the File/Open recent > submenu
- you create a new database with the File/New... menu, and it is opened for you after creation

In all these cases, the framework will prompt you for the password, if it detects that the database is encrypted. It will only open the database if the correct password is given, or if the database is not encrypted.

You will find a detailed explanation of the Data Access Object pattern used by the framework, and how to implement your domain layer in the worked example on page 27.

The steps detailed there lead to having the domain layer available to the rest of your application by implementing the DatabaseOpening facade, and providing an object that implements DatabaseOpeningFacade.

The DatabaseOpeningFacade object must create a subinterface of DAOFactory, and an implementation of it. See the diagram on page 31 for details. Your DAOFactory provides access to additional Data Access Objects that represent the major concerns in your persistence domain. In the case of the worked example these are Accounts and Transactions.

To make your application's DAOFactory available to the rest of your application, the DatabaseOpeningFacade must return both this DAOFactory subinterface, and the implementation of it. Since in Java methods can only return a single value, these are stored in an InstancePair<DAOFactory>. The constructor for this requires its first argument to be class that is a subinterface of DAOFactory, and its second argument must be an instance of your subinterface.

Database closing facade

The framework will close the underlying database for you when you use the File/Close or File/Close All menu items, or when you exit the application.

However, that is a close of the low-level database objects. In your data access layer, you may need to cache some of the database contents and want to release their storage, or you may be using an object-relational mapping system such as Hibernate and need to close your Session.

To hook into the framework to perform such cleanup actions before the underlying database is closed, your plugin must implement the DatabaseClosing facade, and from there provide an object that implements DatabaseClosingFacade.

Since it's likely that such a DatabaseClosingFacade will need access to objects that are created in your DatabaseOpeningFacade, you could implement DatabaseOpeningFacade and DatabaseClosingFacade in the same implementation, and have your plugin implement DatabaseOpening and DatabaseClosing, returning the same instance of your combined facade.

Database migration facade

As described on page 39, the framework records the code version of all plugins, and the version of their database schemas. These values are returned from the getVersion() and getSchemaVersion() methods of Plugin.

Over the lifetime of your application, you may need to change the database schema to accommodate new requirements. When this happens, you ship a new version of your application with changes to its DAO layer etc. that implement these changes. You must also increase the schema version number.

The framework provides a facility to detect that the opened database is at a different version than the running code requires, and can call code you provide in order to effect an upgrade (a.k.a. migration) of the database to the latest schema version.

The current code/schema version values are then stored in the framework's Versions table so that subsequent opening of the database with the same version of the application does not re-trigger a migration.

Version numbers must never decrease. There are two possible scenarios in which this can happen:

- you mistakenly decrease the version number
- you try to open a database that was created by more recent software than you have (e.g. you have version 3, Bob has version 7, and Bob sends you a database he's created)

This will be detected and the framework will refuse to open the database.[7]

Versions can start with a 'v', or not. They are otherwise in the form of numbers interspersed by single full stops, with an optional classifier at the end. Some examples: 1, v1, v1.0, v1.0.2-alpha, 4.5.5565-GA.

The comparison and therefore ordering of versions follows certain rules:

- Versions are right-padded with zero numbers so that both have the same number of numbers (e.g. 2.3.44 has three sets of numbers: 2, 3 and 44). So when comparing 2.3 against 2.3.45.1, 2.3 is right-padded with zero numbers to form 2.3.0.0.
- Then, each set of numbers is considered in turn, so comparing 2.3.44 against 2.3.46, the 2 from both matches so is ignored, then the 3 from each matches so is ignored, finally since 44 < 46, it is determined that 2.3.44 < 2.3.46. In the padding case above, 2.3.0.0 < 2.3.45.1 since the third number comparison gives 0 < 45.
- If a version contains a classifier, such as -alpha, -beta, -GA[8], -FCS[9], this indicates that the version is earlier than a version without a classifier. Ordering of classifiers is not currently supported: it could be argued that there should be an ordering 1.2-alpha < 1.2-beta < 1.2 <

7 Some applications (notably a certain popular office suite) will open the new-format document in a compatibility mode, and should refuse to let you use any of the new features of the application that cannot be saved in the old document format. I feel that this would add extra complexity to application code, and have not yet provided for it. In some cases, the file format has changed completely, and users of the old version will not be able to open new-format documents at all, forcing them to upgrade. Users *should* be upgrading your software regularly, to receive bug fixes, and new features. This is less of a problem for Open Source applications, but in the commercial world, the first user in an organisation to start using the new version would pressurise the organisation to start a potentially costly upgrade cycle.

8 'General Availability'.

9 'First Customer Ship'.

1.2-GA but since the meaning of such classifiers is not universally agreed upon, I've chosen that anything with a classifier is earlier, so release your final products as 1.2, not 1.2-GA or 1.2-FCS[10].

To make your plugin migration-aware, it must implement the DatabaseMigration interface, and via this, provide an implementation of the DatabaseMigrationFacade.

Upon opening a database, its schema version is checked against that returned by your plugin. If a migration is required, the user will be informed, and given the option to continue with the migration or to abandon it. If the user chooses to migrate, your implementation of the DatabaseMigrationFacade's migrateSchema method will be called. Your code will be given low-level objects (DataSource and SimpleJdbcTemplate) with which you may perform any upgrade DDL/DML statements.

The call to migrateSchema will be made inside a transaction. Should your upgrade fail for any reason, you should throw a relevant DataAccessException, and the transaction will be rolled back. If your upgrade succeeds, then just return normally from migrateSchema.[11]

After migrating, the database is opened and available in the rest of your application as discussed in the section on database opening on page 41.

File storage alongside databases

Currently not supported; the intention is to allow applications to store data in separate files in the database's directory, and provide the same encryption facilities for such files as is provided for the database storage.

Presentation

Database creation wizard pages facade

When a user creates a new database, the framework provides a wizard to guide the user through the various stages of creation:

- An introduction page explaining that database files are kept in a directory which you choose, and that they may be optionally encrypted.
- A file chooser which allows you to choose a new, empty directory in which to create the database (or to create a new one, should you need to).
- A choice of whether the database should be encrypted, and if so, to enter, and verify the

10 Normal people understand 'straight' version numbers e.g. 1.2 < 1.3 since it's just decimal numbering. They'd be OK with time-based versions like 2009, 2010, or even 2009.1, 2009.2. They're less sure with 1.3.4 < 1.3.6 since there are three sets of numbers. Knowledgeable users may be aware of -alpha and -beta, but only experts and presumably marketing types would understand -GA and -FCS. If you choose opaque names for your releases, you'll just have confused users: *I* know that 4 < 2000 < XP < Vista < 7 but to normal people, it's very confusing. Why not keep it simple?

11 Note that due to limitations in the current version of the H2 database, which the framework uses, execution of DDL within a transaction is problematic: H2 will not allow it to be rolled back. See http://markmail.org/message/3yd3jxfgia7lzpq5?q=h2+transaction+ddl+list:com%2Egooglegroups%2Eh2-database+from:%22Thomas+Mueller%22

password. Advice on the strength of the entered password is given.[12]

These are the basic wizard pages that the framework provides. Plugins can provide additional wizard pages by implementing the NewDatabaseWizardPages interface, and providing an instance of the NewDatabaseWizardPagesFacade. This allows the plugin to provide a list of WizardPage objects that are appended to the above list of framework-provided pages. Information entered via these wizard pages is stored in a map, and passed to the database creation code. See Database creation facade on page 40 for more information.

For more information on the WizardPage class, consult the documentation for the Netbeans Wizard Framework by Tim Boudreau, available at https://wizard.dev.java.net/

Menu system

The framework is responsible for building and controlling many aspects of your application's menu, with your plugin able to contribute to its customisation. Due to the scope of the framework's menu handling, your application code is NOT allowed to directly manipulate the menu bar, or the framework menus. Your code is given objects which represent the menu, and via these, customisation can be effected.

Here is a breakdown of the purpose of the framework-provided menus:

- the File menu allows
    - new databases to be created via a series of wizards
    - existing ones to be opened either by a file chooser or a 'recent' list; the contents and ordering of items in the 'recent' list is managed by the framework
    - the current open database, or all open databases to be closed[13]
    - the user to exit the application
- the Edit menu is not currently present, but will allow applications to hook into an Undo/Redo system, and a Cut/Copy/Paste system
- the View menu shows a list of different views into your application; these views appear as separate tabs in the main application display
    - some views can be marked as 'permanent', such as the Overview
    - some views are provided by the framework, such as the Overview (the framework provides some 'demo' content for this view; this will be overridden by your application), and the SQL view (a console-based SQL diagnostic tool)
    - some views are provided by your application
- the Tools menu hosts the Options dialog[14]

12 It is not currently possible to:
- enable encryption on a database after is has been created
- change the password if you choose to encrypt
- disable encryption

13 This is not something users *must* do before closing the application: open databases are always closed, but are remembered for the next time the application runs, and automatically re-opened.

14 The intention is that on Mac OS X, the Tools/Options dialog will become the Preferences... dialog under the Apple menu.

- the Window menu shows all active databases and allows the user to switch between them[15]
- the Help menu allows the user to
    - see the welcome screen
    - see a list of new features in this release
    - see the about dialog
    - request a check for update availability

Your application code can also add its own custom menus to the above, as JMenus. Some elements of the above can be customised: the View menu, and (eventually) the Edit menu. Everything else is handled by the framework.

To enable your application to customise the menu, you must implement the MenuProviding interface, and from this, provide an instance of the MenuProvidingFacade. Upon application startup, the MenuProvidingFacade object is given references to several key framework objects to used in customising the menu: the global application menu, the open database list, and a menu facade. These are discussed below.

The ApplicationMenu object(s)

As stated at the start of this section, your application is not allowed direct access to the underlying Swing menu structures. The menu is therefore customised by making changes via an ApplicationMenu object, and the framework provides at least two of these:

- One is for changes to the 'global' application menu, i.e. the menu that's present when no databases are open.
- The other is a 'database specific' application menu, and there is a separate one of these per open database, i.e. this would hold menu items specific to the content of the current database, so in a home finance application, you may have separate entries on the View menu for each bank account in your database.

The global application menu and the database specific application menu are merged to build the actual menu seen by the user.

The ApplicationMenu object allows you to add any custom JMenus that you need. The JMenus are completely up to you to define - the framework will add whatever you give it to the main menu bar.

You may also customise the View menu, but not directly. The menu and view tabs system are related, in that the View menu contains items that allow different view tabs to be made visible or hidden. Some items in the View menu may be hidden until enabled by other parts of the framework or application[16].

View menu items are added to the ApplicationMenu using the addViewMenuTabIdentifier method. To use this, you must create one or more TabIdentifier objects.

15 The framework allows multiple databases to be open simultaneously, with all the views for the database shown in a single window. The Window menu changes the contents of this single window to the current views into the selected database. This is half-way between a single-document interface (too limited) and a multi-document interface (can be confusing). It may change to a 'proper' multiple-document interface in a later release.

16 For example, the SQL view, which may be enabled via the Tools/Options... dialog. Once enabled, a SQL item appears in the View menu, which will show or hide the SQL view.

The TabIdentifier object

TabIdentifiers are the link between menu items on the View menu, and the tabs (views) that you define in your application to show the user differing views into your database, or, if you like, implement all the main screens of your application.

TabIdentifiers define:

- an internal, unique name for the menu item/tab
- a string that will be displayed in the View menu, and in the tab header for this menu/tab
- whether this menu/tab is permanent - if true, its tab will always be shown, and will not be controllable via the View menu; if false, it can be shown and hidden via its entry in the View menu
- a letter for the View menu item's mnemonic
- the name of a bean that will be loaded via the SpringLoader to provide the tab for this. This bean must be an implementation of org.devzendo.minimiser.gui.tab.Tab. There will be more about these Tab objects later.
- an optional parameter stored as an implemention of the TabParameter marker interface that will be made available to the Tab instance when it is loaded. More on this later.

The OpenDatabaseList object

The open database list is a framework object which holds the list of all the currently open databases, knows which one is current (as set via the Window menu) and maintains a list of observers interested in database events.

In order for your application code to make changes to a database-specific application menu, you must create an instance of Observer<DatabaseEvent>, and register it with openDatabaseList.addDatabaseEventObserver().

DatabaseEvents form a small hierarchy. As an observer of DatabaseEvents, your code will be called whenever databases are opened, closed, switched to, or when the last database has been closed. The type of most interest when customising the menu is DatabaseOpenedEvent.

Your observer should check the type of DatabaseEvent, and if it is a DatabaseOpenedEvent, cast it to this type, and extract its DatabaseDescriptor:

```
openDatabaseList.addDatabaseEventObserver(new Observer<DatabaseEvent>() {
    public void eventOccurred(final DatabaseEvent databaseEvent) {
        if (databaseEvent instanceof DatabaseOpenedEvent) {
            final DatabaseOpenedEvent databaseOpenedEvent = (DatabaseOpenedEvent) databaseEvent;
            handleDatabaseOpenedEvent(databaseOpenedEvent);
        }
    }

    private void handleDatabaseOpenedEvent(
            final DatabaseOpenedEvent databaseOpenedEvent) {
        final DatabaseDescriptor databaseDescriptor =
            databaseOpenedEvent.getDatabaseDescriptor();
        final ApplicationMenu databaseApplicationMenu =
            databaseDescriptor.getApplicationMenu();
        populateDatabaseApplicationMenu(databaseApplicationMenu);

    }
```

```
        private void populateDatabaseApplicationMenu(
                final ApplicationMenu databaseApplicationMenu) {
            databaseApplicationMenu.addViewMenuTabIdentifier(
                new TabIdentifier("ACCOUNTS", "Accounts", false, 'A', "accountTab", null));
            menuFacade.rebuildViewMenu();
        }
    });
```

The DatabaseDescriptor object

DatabaseDescriptors hold all information about an open database:

- its name
- the directory where it resides
- its tabbed pane - where the view tabs are shown
- its ApplicationMenu object
- its DAOFactory object

The example code above extracts the database-specific ApplicationMenu object from a DatabaseDescriptor, and adds a View menu TabIdentifier. This will cause the 'accountTab' bean to be loaded.

The MenuFacade object

The menu facade allows your MenuProvidingFacade to trigger a rebuild of the menu, after changes have been made to either the global, or a database-specific application menu. If you have only made changes to a certain part of the menu, you may trigger a rebuild of just that part. Otherwise, the whole menu can be rebuilt.

View system

The view and menu system are related in that the set of tabs visible for the current database is controlled by menu items in the View menu.

Tab lifecycle management

When selected, a menu item on the View menu will cause a new tab to be created in the main tabbed pane for the current database. If selected again, this tab is hidden. The tab bean name is specified in the TabIdentifier. The TabIdentifier may also store a parameter which will be made available to the Tab by the SpringLoader. All tabs are loaded by the SpringLoader in a thread separate from the Swing event dispatch thread (EDT). A tab must be an implementation of org.devzendo.minimiser.gui.tab.Tab.

You may pass any objects you wish into the constructor, including framework objects such as the OpenDatabaseList (bean name 'openDatabaseList'). There are two bean factories whose produce will be of use to you when loading Tabs:

- 'databaseDescriptor' is a bean factory that will be populated with the currently selected database's DatabaseDescriptor before your Tab is loaded; it will be cleared after your Tab loads
- 'tabParameter' is a bean factory that will be populated with the TabIdentifier's TabParameter

subclass instance before your Tab is loaded; it will be cleared after your Tab loads

The way these are intended to be used is as follows:

You register a TabIdentifier with the database-specific ApplicationMenu, as in the example code above. Say you want a separate tab for each bank account in a financial management application. The TabIdentifier may have a TabParameter that encodes the primary key of the account in the database (e.g. 72); it would also use this to form the TabIdentifier's own unique key (e.g. "account72"); it would have read this Account object and used it to form the TabIdentifier's display name (e.g. "Personal savings account"). You may have several TabIdentifiers, one for each account. Each would use the same tab name (e.g. 'accountTab'). This bean might be defined as follows in your application context:

```
<bean id="accountTab"
    class="org.devzendo.beanminder.gui.tab.AccountsTab">
        <constructor-arg ref="databaseDescriptor" />
        <constructor-arg ref="tabParameter" />
<bean>
```

The constructor for this would look like:

```
public AccountsTab(final DatabaseDescriptor databaseDescriptor, final TabParameter accountCode)
```

The constructor would have to extract the account code from the TabParameter.

Should your constructor need to load data from the database, it can extract your application's DAOFactory with:

```
final MyApplicationDAOFactory daoFactory =
    databaseDescriptor.getDAOFactory(MyApplicationDAOFactory.class);
```

The loading of Tabs also takes care of some of the problems present in multi-threaded Swing applications: all graphical components must be created, read and written on the Swing event thread. Therefore, the tab loader will call your Tab's initComponent() method on the EDT, and obtain the main Component via the getComponent() method (on the EDT). Should your graphical initialisation need access to the objects passed into the constructor (or if these have been used to load from the database on the worker thread used to call the constructor), it will be necessary for you to provide adequate synchronization in order to access these safely from the EDT.[17]

Similarly, upon shutdown, disposeComponent() is called on the EDT, then destroy() is called on a non-EDT to free any non-Swing resources.

Status Bar

Dialogs

The framework provides several small standard dialogs as you might find in other applications:

The About and Welcome / What's New dialogs provide details of the application, its license and developer credits, and a list of the features that have been added/improved in this version of the application.

The Problem Reporter dialog is launched when an exception has been caught, and attempts to

17 If you are unfamiliar with threading in Java, consult a good book on the subject: O'Reilly's Java Threads (Oaks & Wong), or Java Concurrency in Practice (Goetz et al) are exceptional.

format the details of the exception in a readable manner, giving details of how to bring this problem to the developers' attention. In later releases, an automated bug submission system may be added to this dialog.

Some of the text displayed by these dialogs is customisable by application plugin developers. The following sections describe these dialogs in more detail, and show how the text they display can be customised.

The About Dialog

Found in the Help menu, the About <Application Name> dialog shows, on three separate tabs:

- the name of the application, a short summary of its license details, and the 'About' text
- the developer credits (these are currently the credits for the framework itself; there is currently no way to customise or add to this, for your application)
- the full license text

All these are customisable by returning the actual text in method calls of Plugin/ApplicationPlugin, or by returning paths to resources in method calls of ApplicationPlugin. These resources must then be present in your application's .jar file – e.g., must be present in src/main/resources, if building the plugin with Maven. These resources can either be text files or (simple) HTML files. They must have .txt or .html/.htm file suffixes.

The “About <Application Name>” menu item shown on the Help menu contains the name of the application as returned by the getName() method of your (Application)Plugin.

The Welcome / What's New Dialog

The Welcome to <Application Name>/ What's New dialog is available from the help menu, and is also shown to you on first startup, to explain the purpose of the application, and to let you browse the recent changes. It also appears if you have upgraded, and then restart the application, so you can find out what has changed (which you may already know, having seen it in the update notification message – more on this later).

Presenting it on first startup does violate one of my GUI guidelines, namely, that applications should not present dialogs without being requested to, as this can be annoying. However, it is important to know what has changed in the application, and it only appears unbidden on install or upgrade once. Also, if users are given an easy mechanism to do something that may be of benefit to them, they may do it, rather than if they have to go and do something out of their way[18]. Hence the two sets of information presented by this dialog are easily switched when the dialog is active, by clicking a button that's present on the dialog. They may of course obtain each of these sets of information by the appropriate menu item – but that would involve some activity on their part, so it might go unused.

The text of the What's New tab is processed from the change log resource. The path to this resource must be returned by the getChangeLogResourcePath() method of the ApplicationPlugin; it must be present on the classpath – e.g. in your ApplicationPlugin's jar. The format of the file is a wiki-like

18 I have no formal Human-Computer Interaction research at hand to back up these claims. The likelihood of the user going out of their way to do something (hunt through menus) depends on their fear of experimentation, their tenacity, how likely the application is to crash if they experiment (which increases their fear, and reduces their willingness to experiment), how valuable the information is that the action would yield.

simple markup style, and is described in the section “The change log”, later in the guide.

Also note that to make use of the *update availability notification* feature, this same file and the latest version number contained therein would be uploaded to your publically-available web server.

The text of the Welcome tab is from the ApplicationPlugin's About resource. The getAboutDetailsResourcePath() method provides the path of this text inside your ApplicationPlugin's jar. The file can have a .txt or .html/.htm suffix, and will be rendered appropriately.

The “Welcome to <Application Name>” menu item shown on the Help menu contains the name of the application as returned by the getName() method of your (Application)Plugin.

The Problem Reporter Dialog

This dialog can be shown when exceptions are caught by your code, as a means of bringing serious problems / bugs to the attention to the user. All the user would be expected to do in this case is copy the description of the problem as shown into an email, and contact the developers.

The email address or URL shown on the problem reporter screen is customisable by the ApplicationPlugin developer by returning it in the getDevelopersContactDetails() method.

To report exceptions to the user, you will either:

- need a reference to the problemReporter bean (obtained from the SpringLoader; in this example, held in the instance variable mProblemReporter), and do: mProblemReporter.reportProblem(“when doing something”, exception) or mProblemReporter.reportProblem(“when doing something”)
- use the static methods ProblemDialogHelper.reportProblem(“when doing something”, exception) or mProblemReporter.reportProblem(“when doing something”)

Problems are reported with a message of the form “A problem occurred ...” followed by the “when doing something” message which you supply. Therefore, please phrase your “when doing something” in terms a user is likely to understand.

Indicating long-running tasks with the CursorManager

Messaging

Messages with “Don't Show This Again” checkboxes

Options or Preferences

Miscellaneous

Logging

The framework uses log4j exclusively; java.util.Logging was introduced after log4j had established a de facto standard, and is unnecessary. Should you wish to use java.util.Logging in your plugin, you will need to provide a suitable adapter between it and log4j – see the slf4j project for the appropriate jar to add as a dependency of your project.

Initialisation of log4j is the first task the framework performs when its main() is called; the command line is used to enable various elements of logging – these command line flags are used to affect the PatternLayout used to customise the log output:

Command line flag	*Pattern changes*
-level	Shows the level of log item, e.g. DEBUG, INFO, WARN, ERROR, FATAL.
-debug	Enables DEBUG output; the default is INFO and above.
-classes	Shows the name of the class generating the log output.
-threads	Shows the name of the thread generating the log output.
-times	Shows the time at which the log output is generated.
-debugall	Turns on all the above.

Normal output would be:

```
Framework starting...
Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@d95da8: display name
[org.springframework.context.support.ClassPathXmlApplicationContext@d95da8]; startup date [Thu Oct 22 08:22:11 BST
2009]; root of context hierarchy
Loading XML bean definitions from class path resource [org/devzendo/minimiser/MiniMiser.xml]
Loading XML bean definitions from class path resource [org/devzendo/minimiser/Menu.xml]
Loading XML bean definitions from class path resource [org/devzendo/minimiser/springloader/SpringLoader.xml]
...
Framework version 0.1.0-SNAPSHOT
Framework name MiniMiser
```

With the -level, -classes, -threads and -times options enabled, output is:

```
INFO  2009-10-22 08:29:04,807 MiniMiser [main] Framework starting...
INFO  2009-10-22 08:29:05,274 ClassPathXmlApplicationContext [main] Refreshing
org.springframework.context.support.ClassPathXmlApplicationContext@358f03: display name
[org.springframework.context.support.ClassPathXmlApplicationContext@358f03]; startup date [Thu Oct 22 08:29:05 BST
2009]; root of context hierarchy
```

```
INFO  2009-10-22 08:29:06,190 XmlBeanDefinitionReader [main] Loading XML bean definitions from class path resource
[org/devzendo/minimiser/MiniMiser.xml]
INFO  2009-10-22 08:29:08,332 XmlBeanDefinitionReader [main] Loading XML bean definitions from class path resource
[org/devzendo/minimiser/Menu.xml]
INFO  2009-10-22 08:29:08,872 XmlBeanDefinitionReader [main] Loading XML bean definitions from class path resource
[org/devzendo/minimiser/springloader/SpringLoader.xml]
INFO  2009-10-22 08:29:10,483 AppDetailsPropertiesLoader [main] Framework version 0.1.0-SNAPSHOT
INFO  2009-10-22 08:29:10,486 AppDetailsPropertiesLoader [main] Framework name MiniMiser
```

Lifecycle management

There is a lifecycle management system available, which can be used to start different parts of your plugin at different phases of the application startup.

WRITE MORE

Event notification

There are also mechanisms whereby your plugin can be called in response to various application events, e.g. databases being created, opening, closing, etc.

Database event notifications

To be notified of databases opening, closing, being switched to, or the last database being closed, register an Observer<DatabaseEvent> on the OpenDatabaseList. This object may be obtained using the 'openDatabaseList' bean name and the SpringLoader. See The OpenDatabaseList object in the Menu section on page 46 for an example of this.

Tab event notifications

Upgrade event notifications

WRITE MORE

The startup queue

WRITE MORE

Background processing with WorkerPool and DelayedExecutor

WRITE MORE

The change log

The change log is a text file that lists all the public releases of your application, together with their release dates, a one-line summary of each release, and any relevant details or new features in each release.

It is intended to be understood by end-users. Developers could use your bug tracking database, such as Bugzilla, Trac, JIRA, etc.

The change log is written in a simple markup language, similar to that which you might use in a

wiki. Certain parts of the change log are understood by the framework, and rendered specially; other parts allow you to impose some formatting conventions such as bullet lists. The markup is quite light, and the change log is still readable as an ASCII text file.

When shown to the user, it is parsed, possibly cut down, and rendered in HTML.

The change log is used in two parts of your application:

- by the “What's new in this release?” dialog, which shows the whole of the change log in its HTML-rendered form. This is the change log for the version of the software that is currently running. It is packaged inside your application's .jar file, and must be called “changelog.txt”, at the root of your package hierarchy – e.g. if building with Maven, it should be in src/main/resources/changelog.txt
- by the update notification system, which retrieves the latest version number and latest change log of your application from your release HTTP server. It then compares the remote, latest version against the version currently running, and if there are any updates, cuts down the remote change log, and sorts by version number to show only the features of more recent releases than the one you are currently running. This subset of the new, exciting updates you could receive with the latest version is then rendered in HTML and presented to the user as a message, using the messaging system.

The change log markup syntax

Example

Here is an example illustrating the features of the change log markup available to you, and the overall style of the changelog.txt file.

```
# comments start with hash/pound/octothorpe
# and are ignored

v1.0 25/06/2009 dawn of a new epoch -
The first release

v0.9.9 not for public consumption

v0.9.8 insert witty title
------
Fixed some bugs
* with a
* simple list
and some text

v0.9.7 : colon separating witty title not out of order
------
* now handles long lines that are so long that they
  stretch onto a separate line but are part of the same bullet and
  so don't get line breaks interspersed.
* and bullet points
```

Structure

Each release is described in its own section, separated by blank lines. Each section has a header line, followed by optional information lines, detailing the features in that release.

Header line structure

The header line must be in the format:

```
version [date] [title-text]
```

Versions can start with a 'v', or not. They are otherwise in the form of numbers interspersed by single full stops, with an optional classifier at the end. Some examples: 1, v1, v1.0, v1.0.2-alpha, 4.5.5565-GA.

Dates are of the form nn/nn/nn or nn/nn/nnnn – the square brackets shown in the header line above are not present in the change log; they indicate that the date is optional. Dates are only validated for their structure as shown – they are not checked for locale conventions, or even whether they are valid dates. Examples (the 25th of December 2009): 12/25/2009, 25/12/2009, 25/12/09, 12/25/09

The title text is also optional, and can provide a one line summary of the release, or a release nickname. Some examples: “world domination is surely ours”, “for demonstration at trade show”, “Winter 2009 Release”.

Optional information line structure

Lines after the header that do not match the header structure are treated as information lines. They are translated into HTML interspersed with line breaks, as they appear in the changelog.txt file.

The only formatting currently supported is the processing of bullet points.

A line starting with an asterisk (*) starts an un-numbered list (<ul> ... ...), thusly:

```
* with a
* simple list
```

Each subsequent line starting with a * adds another item to the list.

Long, multi-line bullet items are achieved by placing the bullet on one line, then placing spaces at the start of subsequent lines, thusly:

```
* now handles lines that are so long that they
  stretch onto a separate line but are part of the same bullet and
  so don't get line breaks interspersed.
* more bullet points
```

The update notification system

Overview

The user has downloaded your application, and is hopefully using it productively, but that's not the end of the story – the community will continue developing features, and making new releases of the application.

Linux users have long enjoyed the operating system's superior package management facilities: where the OS itself notifies the user of updates to applications, and allows them to update their systems.

Mac OS X and Windows users have the same facilities – but only for software distributed by Microsoft and Apple. InstallShield has a service whereby all software installed via InstallShield can be updated from their update application, but not all software is installed via InstallShield. Firefox also has its own update service and ecosystem. The main mechanism for software on Windows and Mac OS X to update itself is for it to implement the update system itself. This is a costly, error-prone approach, and is also wrong. This should be a facility of the operating system. I do not expect this situation to change, since there are companies founded on installers and update systems, and these would probably complain if the OS vendor were to remove their source of income.

Update availability notification

Minimiser does not attempt to provide this automatic update facility. However, the user should be notified that updates to the software exist.

The framework will periodically check with the developer's update site for details of new releases, and notify the user.

On Windows and Mac OS X, the user must then download the software and install it (presumably, uninstalling the previous version first, or if using an intelligent installer, correctly updating the existing version).

On Linux, the in-built update mechanism can be used to deliver the update, providing your repository has been added to the list of Software Repositories. The information provided by the framework's update notification messages – i.e. what's new – should also be published for viewing by the Linux update manager.

Privacy concerns

No user information is sent to the developer's server, and the only information that is retrieved is the latest version number, and, if this is different to that installed (i.e. if there's an update), the change log is retrieved.

As a developer, it might be advantageous to know how many users are using which version of the software – but this information is not sent. If there is sufficient demand for it, this could be implemented.

The transfer takes place over unencrypted HTTP, although the port can be specified. It defaults to port 80.

The IP address of the connecting PC (i.e. where the application is installed) may be logged by the developer's web server.

Some users might view this as a privacy issue, and so, on first startup, they are prompted to confirm whether they want to use the update check, and receive notifications.[19]

Making use of an update site

Your ApplicationPlugin should return a URL from the getUpdateSiteBaseURL() method. For example, http://updates.myapplication.com/latest/

You must be able to publish files from this URL, perhaps it is part of the same server from which

19 At the moment, there is no mechanism to opt-in later; the relevant check-box will be added to the Tools / Options dialog in a later release of the framework.

you make the software downloads available.

The framework will attempt to download two files under this URL:

- version.txt which contains the latest version number on a single line of text, e.g. 'v2.1.3'
- changelog.txt which is your application's latest change log, in the markup style described in the section of this guide entitled “The change log”.

Using the example above, the framework will attempt to first download http://updates.myapplication.com/latest/version.txt and if successful, and there are updates to the current version, it will attempt to download http://updates.myapplication.com/latest/changelog.txt

It will then parse the change log, sort the entries by version, and show only those versions that the user could update to. There is no point showing the user all the versions back to the first release, including the one they are running – the message highlights the changes that the user could be benefiting from.

Periodic update availability checks

The framework will check the remote update site only once per day. This check will only be done if the user has granted the framework the option to make update checks. It will perform the check about an hour after startup, so as to not overly bug the user if they have started the application to Get Work Done Quickly. If the check has been performed today, and the user restarts the application, the check will not be done again until tomorrow.

When the check is being performed, there is no indication to the user. If an update is available, the message button will appear, and the user can read about the update at their leisure.

User-driven update availability checks

The “Check for updates” option on the Help menu is available if the user has granted the framework the option to make update checks.

Since this is user-initiated, not a background task, it gives the user feedback is the check is performed. If an update is available, the message button will appear, and the user can read about the update at their leisure.

Resources

Deployment and Packaging

Overview

The Maven Assembly Plugin can be used easily to gather the transitive dependencies of your application, and all of the framework's dependencies into a directory structure that can be used with some launching mechanism to run your application. This structure would then be packaged into an installable product for the operating system you are targetting:

- For Mac OS X, Apple provide a built-in mechanism for launching Java applications. Applications are typically packaged into .app structures and provided on .dmg disk image files, for easy installation by dragging the application to your system's Applications folder.

- For Windows, applications would be structured with a directory for the jars, and a launcher program. You could ship an executable jar, but that seems to be an unpopular mechanism for launching. You could ship a .cmd script, but this is unattractive since Windows insists on presenting a command prompt window when users launch these graphically. There are several launcher programs available for Windows; launch4j and Janel are examples. Once a launcher structure is available, it is typically wrapped into an installer, using programs such as InstallShield, install4j, Nullsoft NSIS, Microsoft Installer (MSI) etc.
- For Linux, a structure of jars and launcher shell scripts is a typical approach, packaged into a variety of installable formats: .tar.gz, .deb, .rpm, etc.

For Windows and Linux, the set of dependencies is straightforward: just a directory full of jar files.

For Mac OS X, the launcher structure is slightly complicated, since the MiniMiser framework adds dependencies on the Quaqua look and feel jar. This requires the addition of two native libraries, at the same location as the quaqua.jar. These libraries are provided in the DevZendo.org Maven repository at the coordinates ch.randelshofer:libquaqua:6.5:zip.

Structuring cross-platform application projects

Due to the differing launcher directory structures needed for the three main platforms , it is best to structure your application as a Maven multi-module project.

The main, parent project contains the sub-projects for the application core, and operating-system-specific parts (mostly launcher generation).

Platform-specific dependencies can then be isolated to the appropriate project. For instance, the Quaqua native libraries can be declared as dependencies by the Mac OS X sub-project. Eclipse launchers for the Mac OS X variant would then set the java.library.path as being relative to the target directory of the Mac OS X sub-project.

The BeanMinder sample application is structured in this manner; consult the source repositories for further details.

Using the CrossPlatformLauncherPlugin

There are several programs available for creating launchers, as mentioned above, usable from Maven.

As well as the MiniMiser framework, DevZendo.org also hosts a Maven plugin which can build launcher structures for all three common platforms (Mac OS X, Ubuntu Linux and Microsoft Windows).

This plugin can be used with the usual Maven plugins (e.g. Assembly, Dependency plugins) to build launcher directory structures for all three platforms.

The BeanMinder sample project contains configuration in the pom.xml files which makes use of this plugin for the three platforms: consult the project source repositories for details.

Note that the CrossPlatformLauncherPlugin is currently in alpha, and snapshot releases of it are made available in the DevZendo.org Maven repository. The project home page is http://devzendo.org/content/xplp

The plugin generates the correct directory structure containing all dependencies - this can be useful

on Mac OS X, when launching your application. It will extract the Quaqua native libraries under the BeanMinderMacOSX/target/macosx/BeanMinder.app/Contents/Resources/java/lib directory. To reference these in an Eclipse launcher, add -Djava.library.path=${project_loc:BeanMinderMacOSX}/target/macosx/BeanMinder.app/Contents/Resources/java/lib as a VM argument. You will need to run mvn package once before launching this in order to extract all dependencies.

The following sections describe the structure this plugin will create under the target directory.

Mac OS X structure

Created under target/macosx:

project name.app

- Contents
  - MacOS
    - JavaApplicationStub
  - Resources
    - Java
      - lib
        - *all project jar files*
    - *project name*.icns
  - Info.plist
  - PkgInfo

Linux structure

Created under target/linux:

bin

- *project name*.sh

lib

- *all project jar files*

Windows structure

Created under target/windows:

project name.exe

project name.lap

lib

- *all project jar files*

Index