MŰEGYETEM 1782

Budapest University of Technology and Economics
Department of Measurement and Information Systems
Fault Tolerant Systems Research Group

Model Driven Software Development and Service Integration courses

An Overview of the Eclipse Development Environment

Oszkár Semeráth Gábor Szárnyas

August 11, 2014

Contents

Chapter 1

An overview of the Eclipse development environment

1.1 Introduction

The following chapter serves as an introduction to the Eclipse Development Environment. Eclipse is used in both the Model Driven Software Development and Service Integration courses.

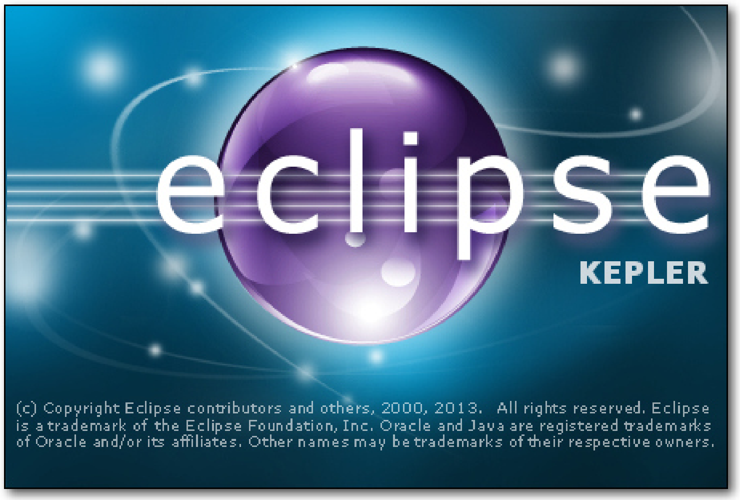

eclipse
KEPLER
(c) Copyright Eclipse contributors and others, 2000, 2013. All rights reserved. Eclipse is a trademark of the Eclipse Foundation, Inc. Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Figure 1.1: The splashscreen of Eclipse Kepler

Eclipse is a free, open-source software development environment and a platform for plug-in development. Members of the Eclipse Foundation include Cisco, IBM, Intel, Google, SAP, Siemens, etc. A list of Eclipse Foundation Members is available here: http://www.eclipse.org/membership/showAllMembers.php.

In this section we will cover the basic concepts in Eclipse.

Eclipse comes in different editions, e.g. Eclipse IDE for Java Developers, Eclipse IDE for C/C++ Developers, Eclipse Modeling Tools, each containing a different set of plug-ins.

The Eclipse SDK includes JDT (Java Development Tools) which features a full-blown Java development environment with an advanced editor and debugger. It supports unit testing (JUnit) and different source code analysis techniques. The JDT has its own Java compiler which can compile Java code incrementally.

Project homepage: http://www.eclipse.org/jdt/

1.2 Project management

1.2.1 Workspace

Eclipse organises the work in *workspaces*. A workspaces can contain multiple *projects*. Projects can be organised in *working sets*.

Upon launching, Eclipse prompts you the location of the workspace. You may switch workspace later in the **File | Switch Workspace** menu.

Further reading:

- http://help.eclipse.org/kepler/topic/org.eclipse.platform.doc.isv/guide/resInt_workspace.htm
- http://help.eclipse.org/kepler/index.jsp?topic=/org.eclipse.platform.doc.user/concepts/cworkset.htm

1.2.2 Project

Eclipse organises the work in *projects*. Projects can have multiple natures, e.g. Java, C++, Plug-in project, etc. You can create a new project in the **File | New** menu.

The project settings are stored in the `.project` file.

Warning: upon creation, the *project directory* will be the same as the *project name*. However, if you rename the project, it only edits the `.project` file and the project directory will not change. To rename the project directory you have to rename it in the file system and import the project. Of course, this is not trivial if you use version control. Thus, when creating projects it's worthy to think on good project names.

Projects can be exported in the **File | Export** menu. A common way of distributing sample projects is to create a zip file by choosing *General | Archive file*. You can import an archive file in the **File | Import** menu by choosing **General | Existing Projects into Workspace** and using the **Select archive file** option.

Further reading: http://help.eclipse.org/kepler/index.jsp?topic=/org.eclipse.platform.doc.isv/guide/resAdv_natures.htm

1.2.3 Package Explorer and Project Explorer

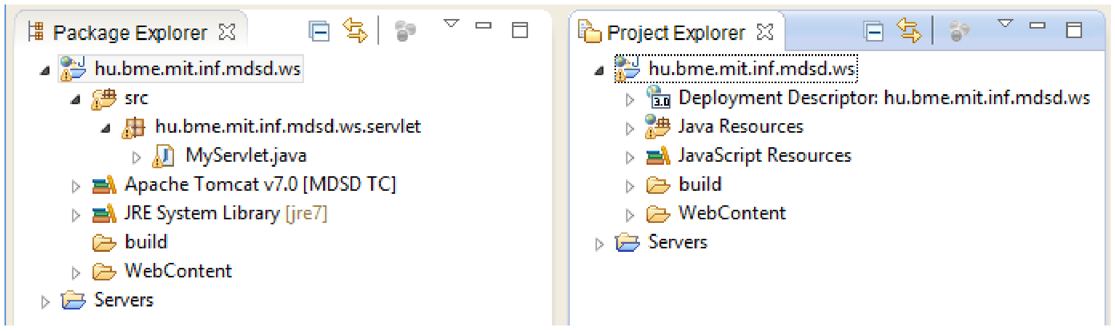


Figure 1.2: The Package Explorer and the Project Explorer in the same workspace

You may want to show the `.project` file in the **Package Explorer**. In order to do so, click the downward pointing triangle in the upper right corner, pick **Filters...** and untick the **.* resources** checkbox.

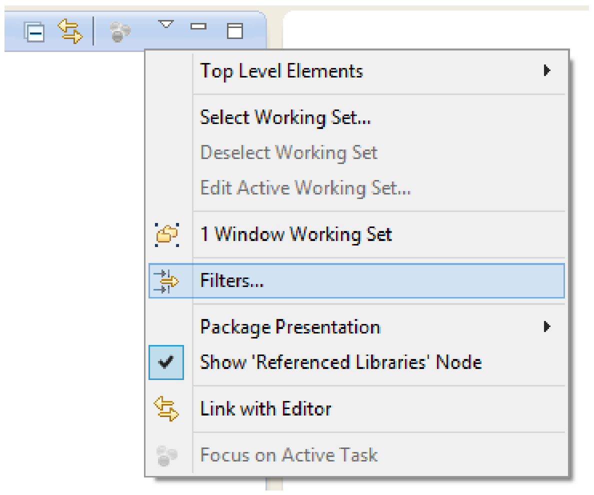


Figure 1.3: The **Filters...** menu in the **Package Explorer**

To show the `.project` file in the **Project Explorer**, click the downward pointing triangle in the upper right corner, pick **Customize View...** and untick the **.* resources** checkbox.

1.2.4 Build in Eclipse

Eclipse's build philosophy is to always keep the source code and the binary code in synch. In order to do so, Eclipse builds the project automatically upon every save operation.

You may turn of the automatic build process by unchecking the **Project | Build Automatically** menu. However, as a general rule you should not turn the automatic build off.

1.2.5 Copying and linking

Naturally it is possible to add another file to an existing project. It can be done by dragging and dropping it to your project. As a result a dialog window will appear that ask if the file should be copied to the workspace or just referenced (and left it in its original place).

In addition to the basic file management this operation is useful for the version control of documents that are edited outside Eclipse. Manual refresh is required if a file changes out of the IDE.

1.2.6 Pictograms

The **Package Explorer** and **Project Explorer** uses a lot of different icons and pictograms. You can find the description if these here: http://help.eclipse.org/kepler/index.jsp?topic=/org.eclipse.jdt.doc.user/reference/ref-icons.htm

1.2.7 Subversion

Apache Subversion (http://subversion.apache.org/), often abbreviated SVN is a widely used open-source version control system.

Eclipse does not include an SVN client by default. You can install the **Subversive** plug-in from your Eclipse distribution's (e.g. Kepler's) update site by following the instructions provided here: http://www.eclipse.org/subversive/.

For basic usage you only need the **Subversive SVN Team Provider** package. Complete the installation and restart Eclipse. Eclise will ask you to install a Subversive Connector. Choose one which is compatible with your SVN server's version, install and restart Eclipse again.

The pictograms of **Subversive** are similar to the one of **Subclipse**: http://stackoverflow.com/questions/3917925/what-do-the-arrow-icons-in-subclipse-mean

1.3 User interface

1.3.1 Workbench

Upon launching, after you choose the workspace location, the workbench window is shown. A workbench window offers *perspectives*. A perspective contains editors, such as the Java Editor and views, such as the Project Explorer.

1.3.2 Editors

Editors contribute buttons to the *global toolbar*. You can have several instances of the same editor, e.g. you can have a dozen Java source files open and edited. You may run different editors at the same time, e.g. you can edit Java and XML files in the same workbench. Editors can be associated with a file name or an extension, and this association can be changed by users.

1.3.3 Views

The primary use of views is to provide navigation of the information in the workbench. You can think of a view as a representation of the data in the workbench. Views have a *local toolbar*. Editors can appear in only one region of the page, whereas views can be moved to any part of the page and minimized as fast views.

The default JDT views include the **Package Explorer**, the **Problems**, the **Console** view and others. You can open new views in the **Window | Show View** menu.

The Problems view and the Error Log view

The **Problems view** shows the warnings and errors in the workspace's projects. In the **Problems** view click on the Downward pointing triangle icon and pick **Show | Errors/warnings on selection**.

The **Error Log** shows the errors occured in Eclipse. It shows the error message, the date and the plug-in that produced the error.

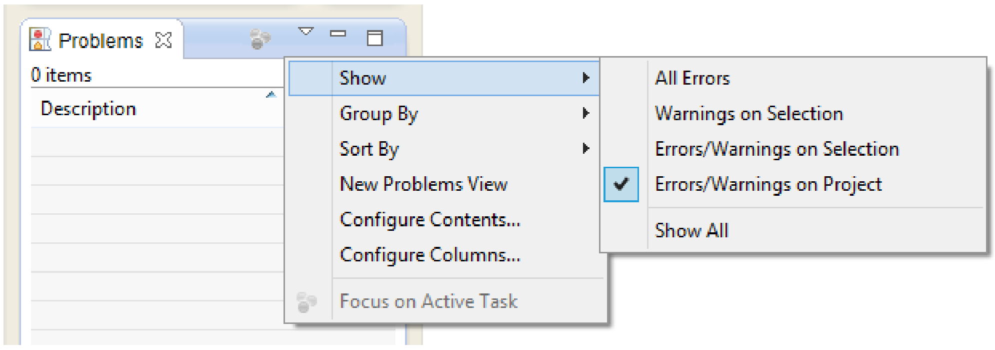


Figure 1.4: Show error and warning on selection

Further reading:

- http://wiki.eclipse.org/FAQ_What_is_a_view%3F
- http://wiki.eclipse.org/FAQ_What_is_the_difference_between_a_view_and_an_editor%3F

1.3.4 Perspective

A perspective is a collection and a predefined layout for editors and views created for a special development purpose: Java, Debug, Plug-in Development, SVN Repository Exploring, etc.

1.3.5 SWT

Java applications typically use AWT (Abstract Window Toolkit) or the Swing toolkit to implement graphical user interfaces. While these are proven solutions, both have serious shortcomings:

- AWT uses native widgets, but only provides the ones which are available on all supported platforms. Also, AWT's architecture implies that the developer has to work on a low level. Hence, AWT is not suitable for modern application development.
- Swing provides its own custom widgets and is easily extensible. Swing provides the option of using either a system „look and feel" which uses the native platform's look and feel, or a cross-platform look and feel that looks the same on all windowing-system. The old Swing implementation suffered from memory consumption and performance problems.

SWT (Standard Widget Toolkit) is a GUI framework that was developed for the Eclipse project by IBM. It uses native components and offers good performance. Today, SWT is maintained by the Eclipse Foundation. Since the SWT implementation is different for each platform, a platform-specific SWT library (JAR file) must be distributed with each application. A number of SWT widgets are available at http://eclipse.org/swt/widgets/.

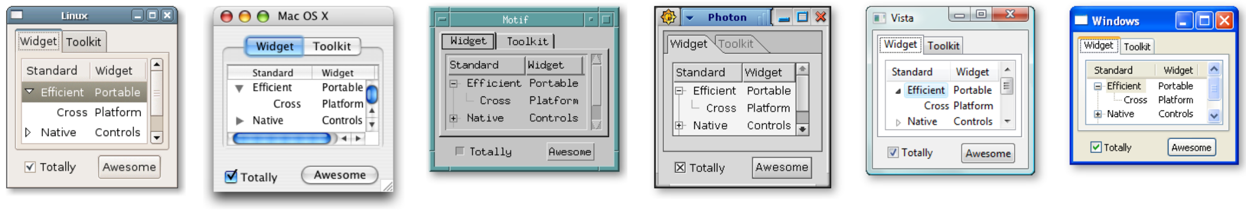


Figure 1.5: SWT widgets on different platforms

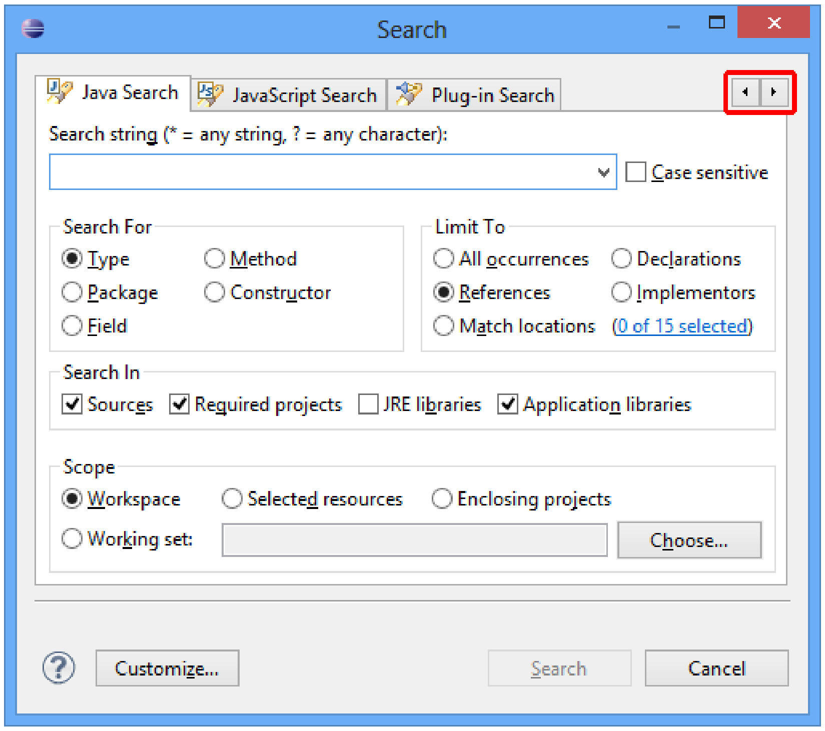
1.3.6 Search

- Search in files: press `Ctrl+H` to display the **Search** window and choose the **File Search** tab. If the window has many tabs, the **File Search** tab may be hidden. The solution is to resize the **Search** window or use the arrows in the upper right corner to show the rest of the tabs.

1.4 Configuration

1.4.1 Bundle

OSGi components are named *bundles*. It's important to note that Eclipse plug-ins are also OSGi bundles.



Search string (* = any string, ? = any character):

☐ Case sensitive

Search For

● Type ○ Method ○ Package ○ Constructor ○ Field

Limit To

○ All occurrences ○ Declarations ● References ○ Implementors ○ Match locations (0 of 15 selected)

Search In

☑ Sources ☑ Required projects ☐ JRE libraries ☑ Application libraries

Scope

● Workspace ○ Selected resources ○ Enclosing projects ○ Working set: Choose...

Figure 1.6: The **File Search** tab may does not appear at first: resize the window or use the arrows

1.4.2 Build path

If a Java project depends on libraries in JAR files, you have to specify the JAR files containing those. In order to do so, you have to add the JAR file to the build path by right clicking on it and picking **Build Path | Add to Build Path**.

By convention, JAR files are typically stored in a directory called `lib`. You cannot add a directory to the build path: you have to specify the files. If you want to remove a JAR from the build path, you have to find it under **Referenced Libraries**, right click and choose **Build Path | Remove from Build Path**.

If you right click anywhere under the project and choose **Build Path | Configure Build Path...**, you can specify the source folders and the requires libaries.

1.4.3 Execution environment

In the **Window | Preferences** dialog click **Java | Installed JREs**. You can add new execution environments and pick the default one.

1.4.4 Run configuration

Eclipse stores the launch settings in *run configurations*. By default, run configurations are only saved in the workspace. If you want to save or share your run configurations, go to the **Run Configurations...** (under the **Run** button or right click on the project and under **Run As**). On the **Common** tab choose **Shared file** in **Save as** group.

If you run multiple programs, you can switch between them by clicking the terminal-shaped icon (called **Display selected console**).

1.4.5 The `.project` file

There is a configuration file in every Eclipse project named `.project`. To make the `.project` file visible from Eclipse, refer to the *Package Explorer and Project Explorer* section. At first this defines the tool set that works with the project by naming the natures applied to them. For example the plug-in projects have the following natures:

```
<natures>
    <nature>org.eclipse.jdt.core.javanature</nature>
    <nature>org.eclipse.pde.PluginNature</nature>
</natures>
```

Secondly it defines the builders that run after every save. In plug-in projects Eclipse builds the Java code, the `MANIFEST.MF` and the `plugin.xml` with the following configuration:

```
<buildSpec>
  <buildCommand>
    <name>org.eclipse.jdt.core.javabuilder</name>
    <arguments>
    </arguments>
  </buildCommand>
  <buildCommand>
    <name>org.eclipse.pde.ManifestBuilder</name>
    <arguments>
    </arguments>
  </buildCommand>
  <buildCommand>
    <name>org.eclipse.pde.SchemaBuilder</name>
    <arguments>
    </arguments>
  </buildCommand>
</buildSpec>
```

1.5 The Java source code editor

Right click the left bar in the source code editor and pick **Show Line Numbers**.

1.5.1 Formatting the source code

In a modern IDE you rarely have to format the code by hand. In Eclipse, right click in the editor and pick **Source | Format**. Hotkey: `Ctrl+Shift+F`.

1.5.2 Refactoring

You often need to rename classes, methods and variables. Doing this by hand is an error-prone method: you may forget to rename some occurences of the renamed item. The *rename refactoring* technique takes care of all occurences of the renamed item. To use it, right click on the renamed item and pick **Refactor | Rename...**. Type the desired name and press `Enter`. Hotkey: `Alt+Shift+R`.

1.5.3 Fixing problems

JDT has a very useful feature called *Quick fix*: if there is an error or warning in the source code, it suggests common ways of fixing it (e.g. if you forgot assign a value to an undefined variable, it will define it). Hotkey: `Ctrl+1`.

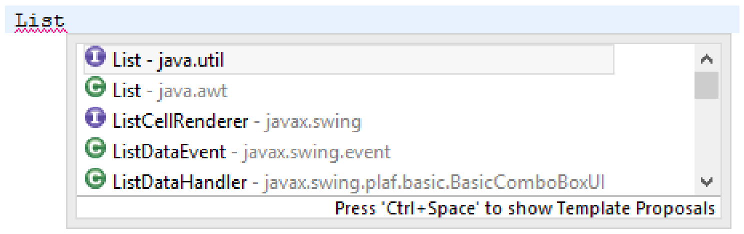
1.5.4 Zooming

By default, Eclipse does not provide zooming in the editor. You can change the font size by going to **Window | Preferences**. Pick **General | Appearance | Colors and Fonts**, and edit **Basic | Text Font**.

1.5.5 Content assist and imports

You can access the content assist by pressing `Ctrl+Space`. Press `Enter` to pick you choice. If you pick an item that has to import a package, the appropriate `import` instructinon will appear between the imports. Sometimes you may end up with lots of unused imports: right click and pick **Source | Organize Imports** or press `Ctrl+Shift+O`.

Pay attention to the package names. For example, the `List` class is available both in `java.awt` and `java.util`.



```
List
List - java.util
List - java.awt
ListCellRenderer - javax.swing
ListDataEvent - javax.swing.event
ListDataHandler - javax.swing.plaf.basic.BasicComboBoxUI
Press 'Ctrl+Space' to show Template Proposals
```

Figure 1.7: Some classes are available in more packages

You can use the content assist by only typing the abbreviation of the desired item. For example, if you have `java.io.InputStreamReader` imported, you can type `ISR` and the content assist will propose `InputStreamReader`.

If you want to overwrite an item (class name, method name) with the content assist, hold the `Ctrl` button when you press `Enter` to pick your choice.

There is a lot of predefined template available in the **Window | Preferences** dialog under **Java | Editor | Templates**. For example, you can type `sysout` to get `system.out.println();`.

You can use templates for control structures, you can define cycles with `for`, `while`, `do`, `foreach` and so on. Similarly, you can define conditional statement with `if`, `ifelse` and `switch`.

Tip: **Organize Imports** can also be used to add missing imports. If the class is available in multiple packages, Eclipse will prompt you to choose between them.

1.5.6 Automatic generation of getter and setter methods

Since the Java language lacks properties, you often have to write getter and setter method for the fields you want to access. Fortunately, you can generate them: right click in the source file and pick **Source | Generate Getters and Setters...**. Similarly, you can generate the constructor, the `toString` method and so on.

If you have only a few properties, there is a quicker way. While typing `getVariableName` or `setVariableName`, use the Content Assist (`Ctrl+Space`), pick the desired method and press `Enter`. The appropriate method is generated.

1.6 Plug-in development

Multiple pre-compiled editions are available at the home page of Eclipse (http://www.eclipse.org/downloads/). One support C/C++ development, an other aids the testing of software. This IDE is not limited to support those popular field of use; it is designed to be as customisable as possible. It can be used for example as a LaTeX editor, as a host of custom enterprise software (e.g. accounting) or even as a note sheet editor (http://mit.bme.hu/~rath/pub/theses/diploma_harmathd.pdf).

So feel free to look for the tools that support your goals.

If you're interested in the topic, the FTSRG has two related courses to offer:

- Eclipse Technologies: http://www.inf.mit.bme.hu/edu/courses/eat
- Eclipse Based Development and Integration: http://www.inf.mit.bme.hu/edu/courses/eafi

1.6.1 Plug-in

Eclipse's main strength is the possibility of creating and installing custom Eclipse plug-ins. Some useful ones are:

- TeXlipse (http://texlipse.sourceforge.net/): „a plugin that adds Latex support to the Eclipse IDE." TeXlipse provides incremental compiling and easy navigation between the TeX source and the generated PDF.
- FindBugs (http://findbugs.sourceforge.net/): „a program which uses static analysis to look for bugs in Java code".
- PMD (http://pmd.sourceforge.net/): „PMD is a source code analyzer. It finds unused variables, empty catch blocks, unnecessary object creation, and so forth."

FindBugs and PMD are widely used tools. They're also part of the *Software Verification Techniques* course (http://www.inf.mit.bme.hu/edu/courses/szet/) of the „Dependable System Design" master's programme held in the autumn semester.

1.6.2 Runtime Eclipse

The Eclipse plug-ins run in an Eclipse instance. If new plug-ins are developed (as Plug-in projects) in an Eclipse instance (called **host Eclipse**), there should be an Eclipse instance that can run them as a part of it in a new empty workspace. This so called **runtime Eclipse** contains the plug-ins installed to the host Eclipse *and* the ones developed in the host Eclipse.

A runtime can be started with the **Run** button. The range of the applied plug-ins can be reduced in the run configuration.

It is possible to install developed plug-in projects (see: Install as a plug-in).

1.6.3 RCP

Eclipse RCP (Rich Client Platform) „is a platform for building and deploying rich client applications. It includes Equinox, a component framework based on the OSGi standard, the ability to deploy native GUI applications to a variety of desktop operating systems, such as Windows, Linux and Mac OSX and an integrated update mechanism for deploying desktop applications from a central server."

Along successful open-source projects, RCP is often used for making highly specialised software for organisations.

Popular RCP applications include the following:

- Java Mission Control (http://docs.oracle.com/javase/7/docs/technotes/guides/jmc/jmc.html): Java Mission Control is a Java profiler provided with the Java Development Kit (JDK) since the release of JDK 7 Update 40.
- ECUTE (http://sourceforge.net/apps/mediawiki/sblim/index.php?title=Ecute): „ECUTE stands for Extensible CIM UML Tooling Environment. It is a family of tools that support all phases of the development of CIM models, CIM providers, and CIM client applications.‘‘ ECUTE is used in the BSc specialisation programme „Information Technologies” on the Intelligent Systems Surveillance (https://www.inf.mit.bme.hu/edu/bsc/irf) course.
- XMind (http://www.xmind.net/): a mind mapping software.
- Vuze, formerly known as Azureus (http://www.vuze.com/): a BitTorrent client.

In the Model Driven Software Development and Service Integration courses we use the following RCP applications:

- Bonita Open Solution: http://bonitasoft.com/
- Yakindu: http://statecharts.org/

For more RCP applications, visit the following links:

- http://www.eclipse.org/community/rcpos.php
- http://www.eclipse.org/community/rcpcp.php

The popular UML and BPMN modelling tool, Visual Paradigm can also be integrated to Eclipse: http://www.visual-paradigm.com/product/vpuml/provides/ideintegration.jsp.

Further reading: http://www.eclipse.org/home/categories/rcp.php.

1.6.4 Update site

Update sites are used to install new features to the Eclipse application. You can install new applications in the **Help | Install New Software...** menu by selecting the update site and the installed components.

After the installation completes, it prompts you to restart Eclipse. If you don't want to restart yet, you can restart Eclipse later by clicking **File | Restart**.

1.6.5 Install as a plug-in

Tutorial: http://www.vogella.com/articles/EclipsePlugIn/article.html#deployplugin_direct

1.6.6 The `Manifest.MF` file

The plug-in project contains a folder named `META-INF`. This folder has a file named `MANIFEST.MF` that describes the relations of the packages of this project with the other packages. In simple words, it defines which (Java) package is visible as you edit the source files in this project, and which package you want to make visible to other projects. The content this file looks like the following:

- Version numbers of the `MANIFEST.MF`:

  ```
  Manifest-Version: 1.0
  Bundle-ManifestVersion: 2
  ```

- Names and versions. This can be edited at the **Overview** page of the in the **Plug-in Editor**. An example content:

```
Bundle-ManifestVersion: 2
Bundle-Name: JPADataCompileButton
Bundle-SymbolicName: hu.bme.mit.mdsd.generatebutton;singleton:=true
Bundle-Version: 1.0.0.qualifier
```

- Required target platform that can run this bundle:

  ```
  Bundle-RequiredExecutionEnvironment: JavaSE-1.6
  ```

- After this, the configuration file enumerates the required bundles with optional minimal version requirement. This can be edited in the **Dependencies** page of the **Plug-in Editor**. An example:

  ```
  Require-Bundle: org.eclipse.ui,
   hu.bme.mit.mdsd.erdiagram;bundle-version="1.0.0",
   hu.bme.mit.mdsd.codegenerator;bundle-version="1.0.0",
   org.eclipse.core.runtime;bundle-version="3.8.0"
  ```

- The following section declares the exported packages. This can be edited at the **Runtime** page of the Plug-in editor.

  ```
  Export-Package: ERDiagram,
   ERDiagram.impl,
   ERDiagram.util
  ```

1.6.7 The `plugin.xml` file

An eclipse plug-in is an OSGi bundle that (usually) connects to other bundles through an extension point mechanism. An **extension point** defines the interface and an **extension** defines a subscription to that arbitrary interface. The `plugin.xml` configuration file contains those information. This configuration can be edited in the **Extension** page of the plug-in editor.

An example subscription that handles a GUI event:

```xml
<!-- This is an extension of "org.eclipse.ui.handlers". -->
<extension
   point="org.eclipse.ui.handlers">
   <!-- If a command named "hu.bme.mit.JPADataCompileButton.GenerateCommand"
      fires then call the execute method of the class named
      "hu.bme.mit.JPADataCompileButton.GenerateCommand".-->
   <handler
      class="hu.bme.mit.compilecommandhandler.JPADataGenerateCommandHandler"
      commandId="hu.bme.mit.JPADataCompileButton.GenerateCommand">
   </handler>
</extension>
```

1.7 Hotkeys

As every modern IDE, Eclipse defines a great number of hotkeys. We gathered some useful ones here:

- List the available hotkeys: `Ctrl+Shift+L`.
- Quick fix: `Ctrl+1`

- Content assist: `Ctrl+Space`
- Organize imports: `Ctrl+Shift+O`
- Autoformatting of source code: `Ctrl+Shift+F`
- Run: `Ctrl+F11`
- Navigate between tabs: `Ctrl+Page Up`, `Ctrl+Page Down`
- Rename refactoring: `Alt+Shift+R`
- Find/Replace: `Ctrl+F`, use `Ctrl+K` to iterate through the results.
- Seach: `Ctrl+H`

You can edit the hotkeys in the **Window | Preferences** menu, in **General | Keys**. For some plug-ins (e.g. TeXlipse), the hotkeys don't appear at first: click the **Filters...** button and untick the **Filter uncategorized commands** checkbox.

1.8 Sources

- Our own experience from Project Laboratories, etc.
- University courses: Eclipse Technologies (http://www.inf.mit.bme.hu/edu/courses/eat), Eclipse Based Development and Integration (http://www.inf.mit.bme.hu/edu/courses/eafi), Model Driven Software Development (http://www.inf.mit.bme.hu/edu/courses/mdsd), Service Integration (http://www.inf.mit.bme.hu/edu/courses/szolgint)
- http://www.eclipse.org/documentation/
- http://theshyam.com/2009/07/eclipse-productivity-shortcuts/
- http://www.openlogic.com/wazi/bid/221090/Eclipse-productivity-tips
- http://rithus.com/eclipse-productivity-tips

Chapter 2

Eclipse laboratory: step-by-step instructions

2.1 Introduction

We will demonstrate Eclipse on a simple task. We create a Java project and then extend it to a plug-in project. After that, we import a plug-in project which puts a button on the Eclipse toolbar and configure it to print the output of our own plug-in project.

2.2 Java project

Go to **File | New | Other...**. Here you can choose any type of projects your Eclipse supports. For now, create **Java Project**.

1. Name to project to `hu.bme.mit.inf.carhandler` and click **Finish**.
2. Right click on the project name and choose **New | Package**. Name the package to `hu.bme.mit.inf.cars`.
3. Right click the package and choose **New | Class**. Name the class to `Car`.

   ```
   public class Car {
        private String numberPlate;
        private int yearOfManufacture;
        private double acceleration;
   }
   ```

   Right click and go to the **Source** menu. You can access formatting, refactoring and generation tools here. Use the following:

   - **Generate Constructor using Fields...**
   - **Generate Getters and Setters...**
   - **Generate toString...**
   - **Format**

4. Create a new class named `CarFleetPrinter`. This time, tick `public static void main(String[] args)` checkbox so the main function is generated automatically. You can create the main method later as well, using the `main` template and content assist (`Ctrl+Space`).

   Write the following code. Hints:

- Type /** and press Enter to Javadoc.
- Use Ctrl+Space (content assist) or Ctrl+1 (quick fix) to use the appropriate package to import.
- Type LL and use the content assist to find the LinkedList class.
- Use Ctrl+1 or Ctrl+2, L to create a local variable (random) for the new Random instance.
- Use the foreach and the sysout templates.
- In the **Window | Preferences** dialog, go to **Java | Editor | Typing**. In the **Automatically insert at correct position** group, check the **Semicolons** checkbox.

```
/**
* Car dealer program.
* @param args Arguments
*/
public static void main(String[] args) {
  String manifest = "The car fleet consists of:\n";

  List<Car> cars = new LinkedList<>();
  Random random = new Random();
  for (int i = 0; i < 10; i++)
    cars.add(
      new Car("MIT-" +
        (String.format("%03d", random.nextInt(1000))),
         2000 + random.nextInt(14),
         3.0 + random.nextDouble() * 4)
      );
    );

  for (Car car : cars) {
    if (car.getAcceleration() < 5) {
      manifest += "- Car: " + car + "\n";
    }
  }

  System.out.println(manifest);
}
```

5. Run the application. Right click the project and choose **Run As | Java Application**. Hotkey: Alt+Shift+X, J.

6. If you would like to read a JRE method's source code, hold Ctrl and click on the class' name. Click the **Attach Source...** button and set the C:/Program Files/Java/jdk1.7.0_51/src.zip.

7. Use the **Rename** refactoring technique to rename the class from Car to SportsCar.

8. Select the following part:

```
return new Car("MIT-" +
    (String.format("%03d", random.nextInt(1000))),
    2000 + random.nextInt(14),
    3.0 + random.nextDouble() * 4);
```

   Use the **Extract method** refactoring technique to extract it to a method named generateCar.

9. Select the whole main method except the last line (System.out.println(manifest)). Use the **Extract method** technique again to extract it to a method named getCarManifest.

10. The result looks like this:

```
public static void main(String[] args) {
  String manifest = getCarManifest();

  System.out.println(manifest);
}

public static String getCarManifest() {
  String manifest = "The car fleet consists of:\n";

  List<Car> cars = new LinkedList<Car>();
  Random random = new Random();
  for (int i = 0; i < 10; i++)
    cars.add(generateCar(random));

  for (Car car : cars) {
    manifest += "- " + car + "\n";
  }
  return manifest;
}

private static Car generateCar(Random random) {
  return new Car("MIT-" +
    (String.format("%03d", random.nextInt(1000))),
    2000 + random.nextInt(14),
    3.0 + random.nextDouble() * 4);
  }
}
```

11. Run the application.

2.3 Jetty

Figure 2.1: The logo of Jetty

1. Create a new Java project `hu.bme.mit.inf.carserver`.
2. Search for “Jetty” and go to http://download.eclipse.org/jetty/stable-9/dist/. Download `jetty-distribution-9.1.2.v20140210.zip`.
3. Search for “Jetty tutorial” and go to http://wiki.eclipse.org/Jetty/Tutorial/Jetty_HelloWorld.
4. Create a `HelloWorld` class and copy the code (with the imports) to it:

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.ServletException;

import java.io.IOException;
import org.eclipse.jetty.server.Server;
import org.eclipse.jetty.server.Request;
import org.eclipse.jetty.server.handler.AbstractHandler;

public class HelloWorld extends AbstractHandler
{
    public void handle(String target,
                       Request baseRequest,
                       HttpServletRequest request,
                       HttpServletResponse response)
        throws IOException, ServletException
    {
        response.setContentType("text/html;charset=utf-8");
        response.setStatus(HttpServletResponse.SC_OK);
        baseRequest.setHandled(true);
        response.getWriter().println("<h1>Hello World</h1>");
    }

    public static void main(String[] args) throws Exception
    {
        Server server = new Server(8080);
        server.setHandler(new HelloWorld());

        server.start();
        server.join();
    }
}
```

5. Change the response to the following:

```
response.getWriter().println(CarHandler.getCarManifest());
```

6. Run in **Debug mode**. Visit http://localhost:8080/ in the browser.

7. The output is a bit messy. Add preformatted text tags to the command:

```
response.getWriter().println("<pre>" + CarHandler.getCarManifest() + "</pre>");
```

8. Refresh the page in the browser.

9. Remove `jetty-http-9.1.2.v20140210.jar` from the build path. The application will remaing free of compile errors. Re-run the application.

10. It will throw `java.lang.NoClassDefFoundError` caused by a `java.lang.ClassNotFoundException`.

```
Exception in thread "main" java.lang.NoClassDefFoundError: org/eclipse/jetty/http/HttpField
  at hu.bme.mit.inf.carserver.HelloWorld.main(HelloWorld.java:31)
Caused by: java.lang.ClassNotFoundException: org.eclipse.jetty.http.HttpField
  at java.net.URLClassLoader$1.run(Unknown Source)
  at java.net.URLClassLoader$1.run(Unknown Source)
  at java.security.AccessController.doPrivileged(Native Method)
  at java.net.URLClassLoader.findClass(Unknown Source)
  at java.lang.ClassLoader.loadClass(Unknown Source)
  at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)
  at java.lang.ClassLoader.loadClass(Unknown Source)
  ... 1 more
```

11. The reason is that the `jetty-http` file is accessedd *transitively*.

2.4 Maven

maven

Figure 2.2: The logo of Maven

1. Use **Maven** instead.

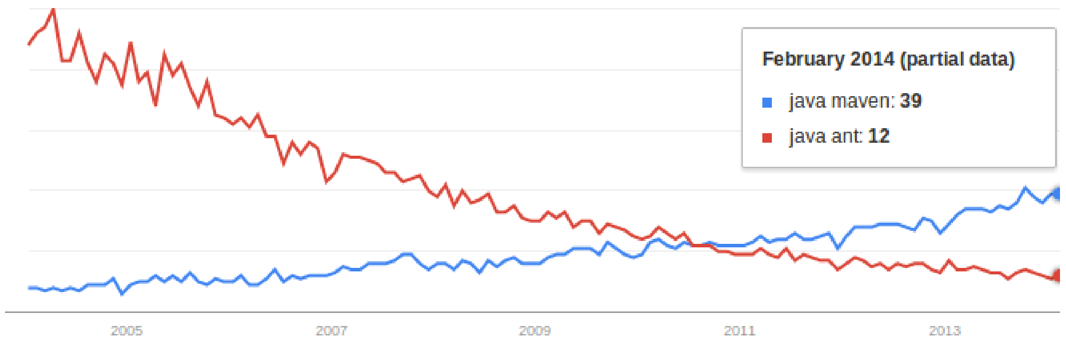


Figure 2.3: Maven and Ant trends

1. Right click the `carserver` project, choose **Configure | Convert to Maven Project...**. Click **Finish**.
2. Go to the **Maven Central Repository** (http://search.maven.org/). Search for `jetty-servlet`. Choose the `org.eclipse.jetty` group's latest `jetty-servlet` artifact. Copy the following Maven dependency to the `pom.xml` file. Do not forget to add the surrounding `<dependencies>` tag.

   ```
   <dependency>
       <groupId>org.eclipse.jetty</groupId>
       <artifactId>jetty-servlet</artifactId>
       <version>9.1.2.v20140210</version>
   </dependency>
   ```

3. If you have the command-line version of Maven installed, you can go to the command line in `hu.bme.mit.inf.carserver` and run:

   ```
   mvn exec:java -Dexec.mainClass=hu.bme.mit.inf.carserver.HelloWorld
   ```

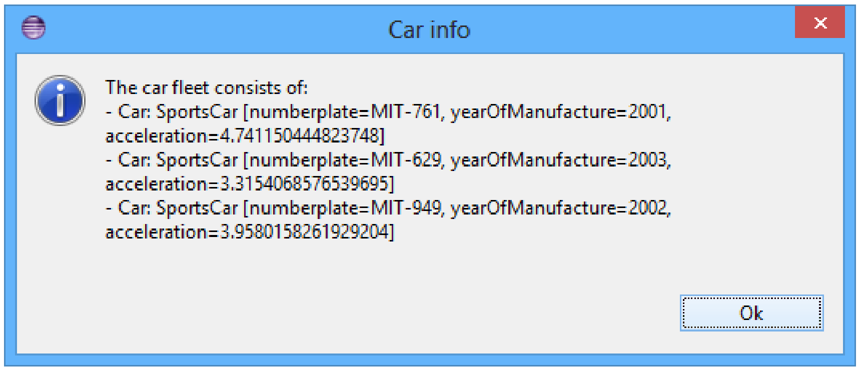
2.5 Plug-in project

1. Import the `hu.bme.mit.inf.car.carbutton` project from the `CarButton.zip` file. Use the **File | Import** menu and choose **General | Existing Projects into Workspace** and use the **Select archive file** option.

2. Inspect the `plugin.xml` and `META-INF/MANIFEST.MF` file. The most interesting for now are the **Extensions** tab.
3. Run the project by right clicking the project and picking **Run As | Eclipse Application**. A new Eclipse instance called „Runtime Eclipse" will start with the plug-in. Close the welcome Window. Observe the **Print!** button on the toolbar.
4. Close the Runtime Eclipse.
5. We would like to extend our Java project to a plug-in project. Right click the `hu.bme.mit.inf.carhandler` project and choose **Configure | Convert to Plug-in Projects...**. Click **Finish**.
6. Go to the `carhandler` project's newly created `MANIFEST.MF` file. Pick the **Runtime** tab. Observe the **Exported Packages**. Later, you can add additional packages if necessary.
7. Go to the `carbutton` project's `MANIFEST.MF` file. Pick the **Dependencies** tab.
8. Click **Add**. An empty list will show. However, as soon as you start typing `car`, the `hu.bme.mit.inf.carhandle (1.0.0.qualifier)` package will show. Click **OK**.
9. Go to the `hu.bme.mit.inf.car.carbutton` package's `PrintTheCarHandler` file. Inspect the `showMessage` method which shows a message in a dialog window.
10. In the `execute` method use `showMessage` to show the car fleet's data.

    ```
    showMessage(CarFleetPrinter.getCarManifest());
    ```

    Use **Quick Fix** take care of the missing `CarFleetPrinter` import and change the visibility of the method.
11. Run the plug-in project. This time, the **Print!** button will show the cars in the fleet.



```
The car fleet consists of:
- Car: SportsCar [numberplate=MIT-761, yearOfManufacture=2001,
acceleration=4.741150444823748]
- Car: SportsCar [numberplate=MIT-629, yearOfManufacture=2003,
acceleration=3.3154068576539695]
- Car: SportsCar [numberplate=MIT-949, yearOfManufacture=2002,
acceleration=3.9580158261929204]
```

Figure 2.4: The message shown in the plug-in project

2.6 Version control

From the **Window | Open Perspective | Other...** pick the **SVN Repository Exploring** perspective. Use the green plus icon (**New Repository Location**) to add a new repository. (You can also import from the **File | Import** menu with the **SVN | Project from SVN** option.) & Specify the URL and fill the authentication data appropriately. If you're working on a private computer, it's recommended to save the authentication data.

Click **Finish**. For now, don't bother with the password recovery feature.

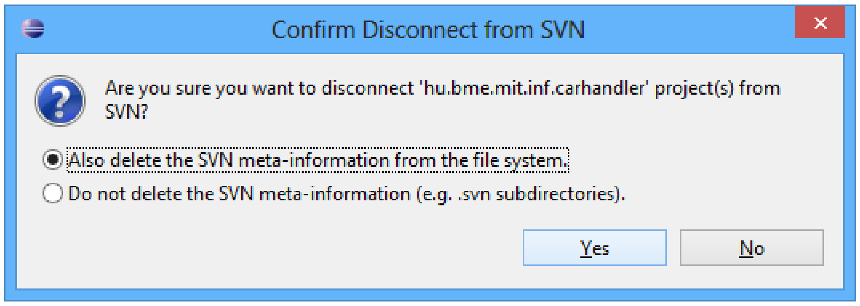
2.6.1 Sharing projects

If you have configured a Subversion repository, you can easily share your projects. Right click on the project name and pick **Team | Share Project...**.

Choose **SVN** and choose your repository location and specify the target URL. Pay attention to always include the project name as the last directory in the path. (Warning: if you use the **Browse...** button, it will not be added automatically).

Click **Finish**. In the **Commit** window fill the commit message and click **OK**.

If you ever decide to stop using version control for a project (e.g. your version tracking got messed up), go to right click menu and choose **Team | Disconnect**. When Eclipse prompts you to confirm the question, choose the **Also delete the SVN meta-information from the file system.** which deletes the hidden `.svn` directories.



Confirm Disconnect from SVN

Are you sure you want to disconnect 'hu.bme.mit.inf.carhandler' project(s) from SVN?

● Also delete the SVN meta-information from the file system.
○ Do not delete the SVN meta-information (e.g. .svn subdirectories).

Figure 2.5: Use the first option when disconnecting from SVN

You can commit files by choosing **Team | Commit...**, update files by **Update**. You can also make good use of the **Revert** and **Revert to commit** options.

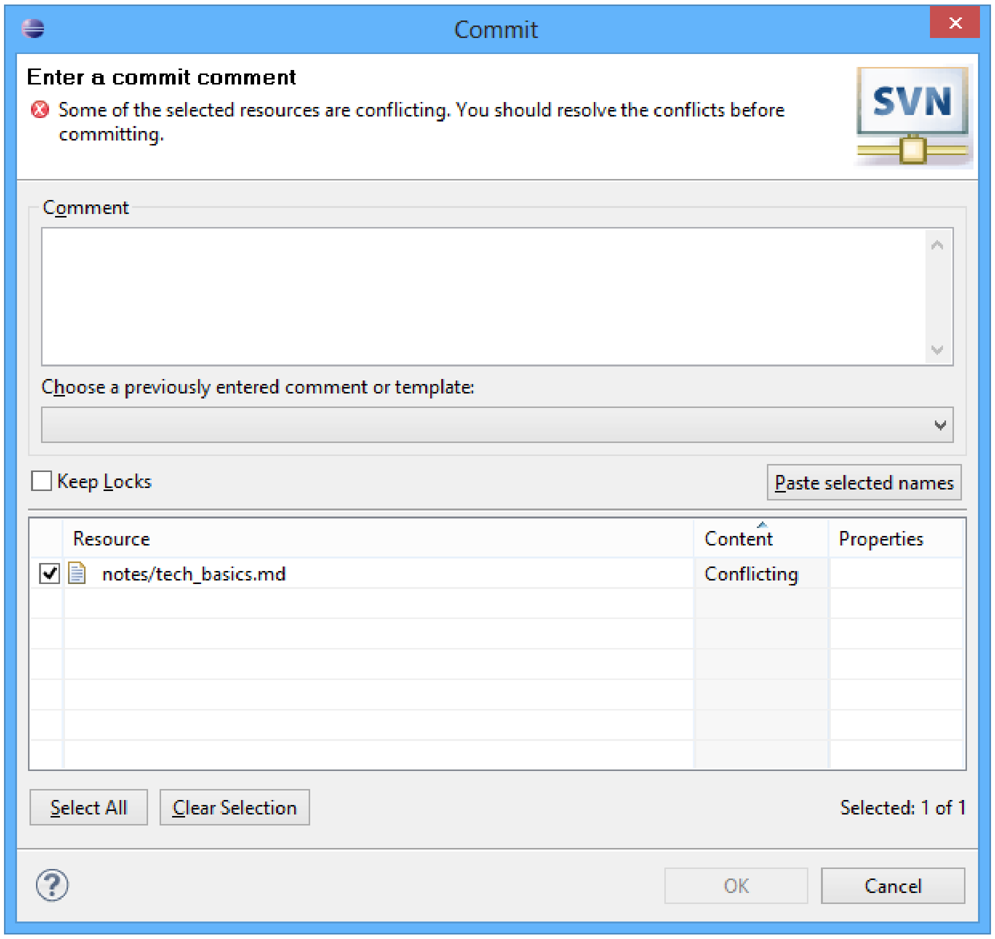
If more than one person works on a file, a conflict can emerge.

To resolve the conflict, use the **Team Synchonizing** perspective or right click on the file and choose **Team | Edit Conflicts**.

Further reading: http://help.eclipse.org/kepler/index.jsp?topic=/org.eclipse.platform.doc.user/tasks/tasks-115.htm

2.7 References

Great tutorial about the configuration of the Eclipse IDE: http://www.vogella.com/tutorials/Eclipse/article.html



Enter a commit comment

Some of the selected resources are conflicting. You should resolve the conflicts before committing.

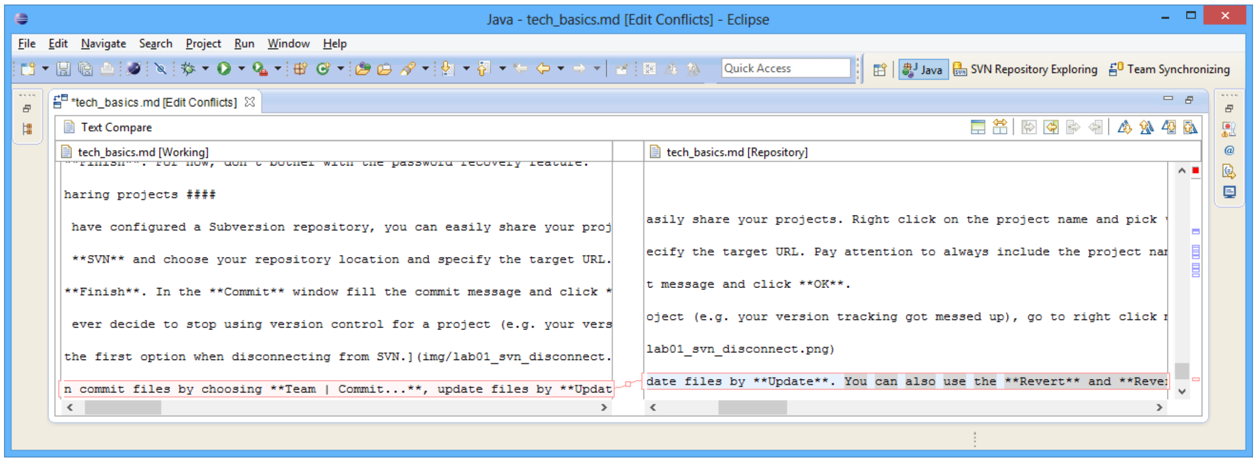
Comment

Choose a previously entered comment or template:

	Resource	Content	Properties
✓	notes/tech_basics.md	Conflicting	

Selected: 1 of 1

Figure 2.6: Conflicting resources



tech_basics.md [Working]

```
haring projects ####

 have configured a Subversion repository, you can easily share your proj

 **SVN** and choose your repository location and specify the target URL.

**Finish**. In the **Commit** window fill the commit message and click *

 ever decide to stop using version control for a project (e.g. your vers

the first option when disconnecting from SVN.](img/lab01_svn_disconnect.

n commit files by choosing **Team | Commit...**, update files by **Updat
```

tech_basics.md [Repository]

```
asily share your projects. Right click on the project name and pick

ecify the target URL. Pay attention to always include the project nar

t message and click **OK**.

oject (e.g. your version tracking got messed up), go to right click r

lab01_svn_disconnect.png)

date files by **Update**. You can also use the **Revert** and **Rever
```

Figure 2.7: Resolving the conflict in the **Team Synchronizing** perspective