BlueJ Teamwork Tutorial

Version 2.0

for BlueJ Version 2.5.0 (and 2.2.x)

Bruce Quig,
Davin McCall

School of Engineering & IT,
Deakin University

Contents

1 Overview

The BlueJ Teamwork features aim to provide tool support for students to be able to work in small teams on programming tasks, projects and assignments. This has been implemented by providing support for a subset of the version control functionality of standard version control systems. BlueJ's teamwork features do not expose students to the full variety of capabilities that version control systems such as CVS and Subversion possess.

A carefully chosen subset of capabilities is supported, allowing students to work together on projects. It supports the main teamwork activities of:

- sharing projects by placing them in a central repository,
- checking out copies of projects from the repository,
- updating local copies,
- committing local changes to the repository version,
- handling merge conflicts,
- providing project status queries,
- and viewing the project history/log.

The teamwork features are implemented over the top of two existing systems, the Concurrent Versioning System (CVS) and (since BlueJ 2.5.0) Subversion. Our aim is to expose students to general version control concepts without exposing them specifically to the inner workings of CVS or Subversion any more than is necessary.

2 Setting up a repository

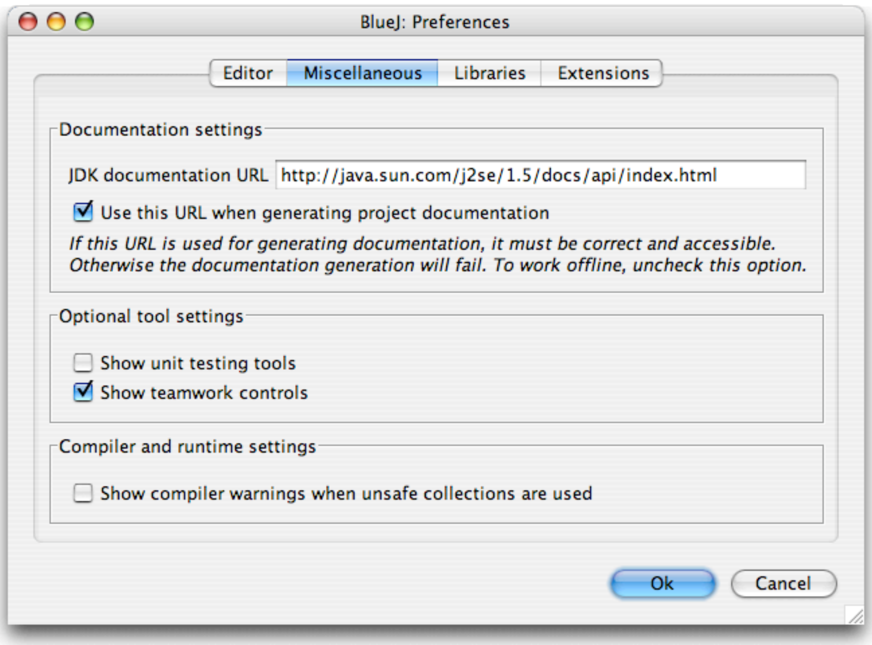
One of the key requirements needed to use BlueJ's Teamwork features is a correctly configured CVS or Subversion repository. There are many ways in which such a repository can be configured successfully.

Information on how to set up repositories, either simple ones for single user projects, or access-protected repositories for groups projects, is provided in a separate document (*BlueJ Teamwork Repository Configuration*).

Unless you have a suitable repository correctly configured and accessible, it is not possible to use BlueJ's Teamwork features.

3 Enabling teamwork features

To help keep BlueJ's user interface as clean and simple as possible, the teamwork features are hidden by default. To enable them, a user should choose the *Tools>Preferences…* menu option (*BlueJ> Preferences...* on Mac OS X). This will show BlueJ's Preferences Dialog (Figure 1). Under the *Miscellaneous* tab there is a check box labelled *Show teamwork controls* which needs to be checked.


BlueJ: Preferences

Documentation settings

JDK documentation URL http://java.sun.com/j2se/1.5/docs/api/index.html

☑ Use this URL when generating project documentation

If this URL is used for generating documentation, it must be correct and accessible. Otherwise the documentation generation will fail. To work offline, uncheck this option.

Optional tool settings

☐ Show unit testing tools

☑ Show teamwork controls

Compiler and runtime settings

☐ Show compiler warnings when unsafe collections are used

Figure 1 BlueJ Preferences Dialog: Show teamwork controls

Upon pressing *Ok* the Teamwork features should now be visible. In the main BlueJ window there will be three additional buttons (*Update*, *Commit* and *Status*) which are disabled for any project which is not currently being shared (Figure 2).

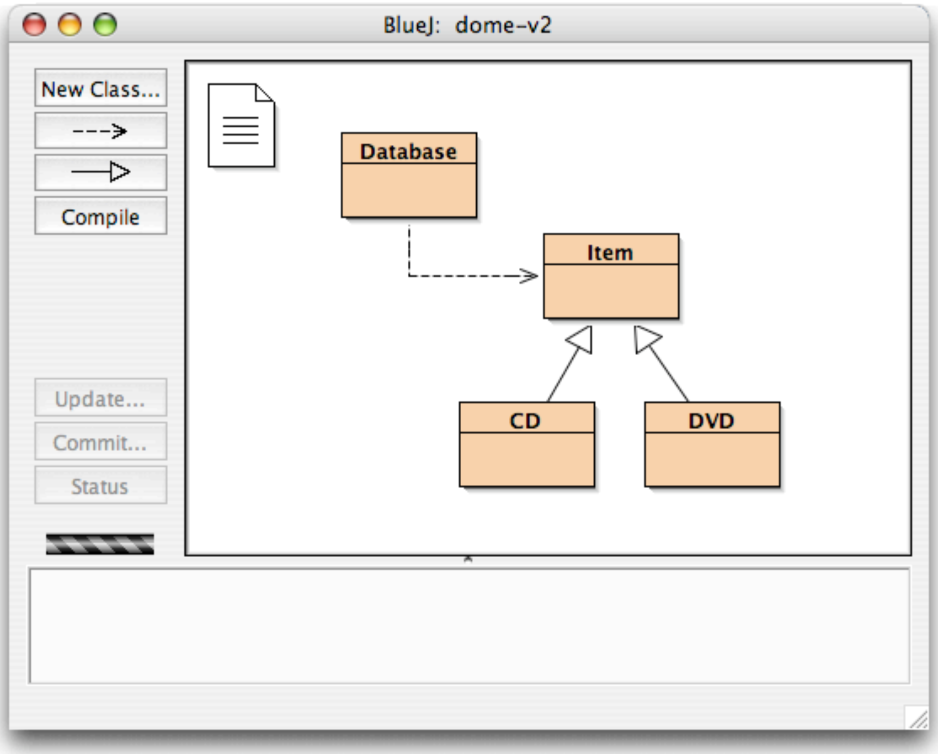


Figure 2 Greyed out teamwork controls in main BlueJ window

There is also an additional Team sub-menu under the Tools menu, which provides some additional Teamwork actions (Figure 3). Two of these will be enabled: *Share this Project* and *Checkout Project*.

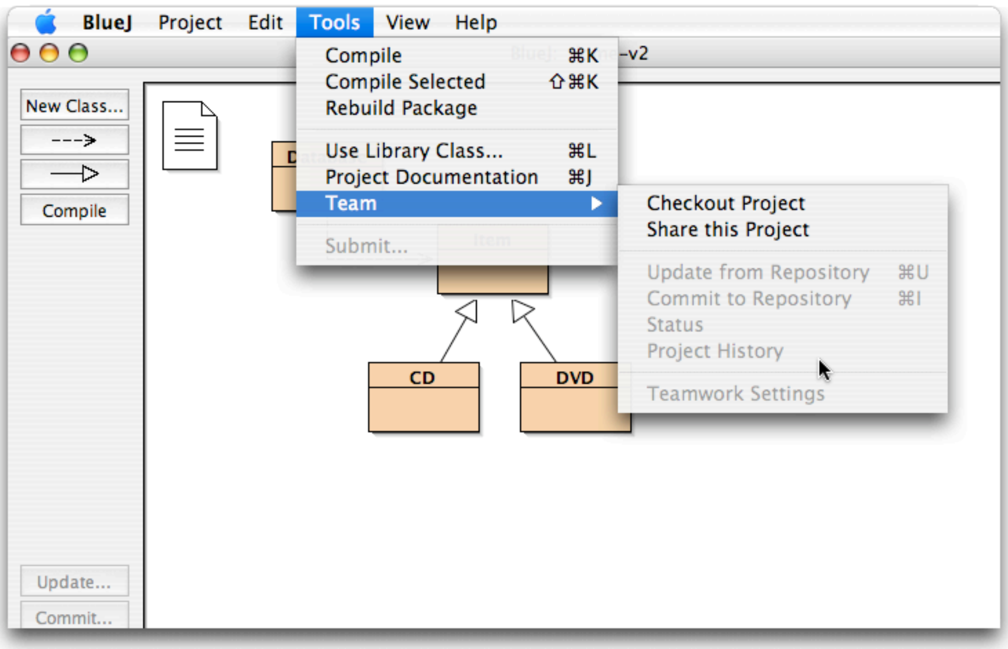


Figure 3 Teamwork menu options

4 Sharing a project

The starting point for interacting with BlueJ's Teamwork features for most users will be one of the two menu options mentioned above. These allow you to share an existing local project (which will place it into a shared repository), or to checkout a copy of a project already in a repository. In this section we describe how to share an existing project. You will need to know the details of the repository that you want to place the project in.

Open the existing local BlueJ project that you would like to share (by placing into an existing repository). Select the *Tools>Team>Share this Project* menu option. At this point you will see a *Team Settings* dialog that requires information about the destination repository (Figure 4).

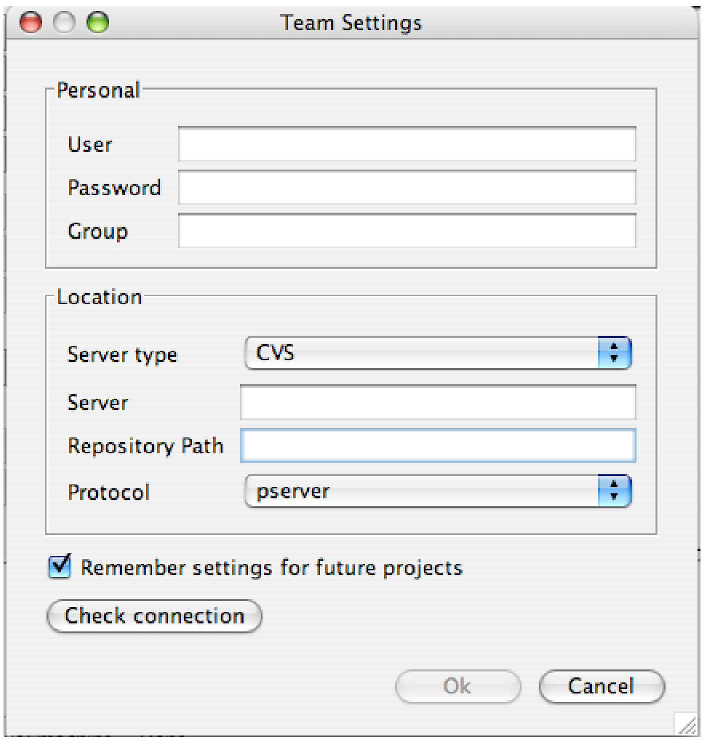


Figure 4 Team Settings Dialog

The following information is required:

User

This is your username that is required for access to the repository. This will typically be your login name for the machine on which the repository is located.

Password

This is the associated password for the username specified above to allow repository access. At various stages when using the BlueJ Teamwork features you will be asked to provide a password for the repository before Teamwork operations are undertaken. These passwords are not permanently stored, and are asked for whenever a Teamwork operation is invoked upon a newly opened project. Once you have supplied this information it is remembered for any further operations during that session.

Group (optional)

Group is an optional field. It is not needed if you are only trying out BlueJ Teamwork features, or you are interacting with a single repository. It is a field that can be used by teachers or system administrators to allow for the specification of separate repositories for each student team. The group name is used to append another directory to the location of the repository path. This can be left blank if not required.

Server type

This is the type of version control server software – as of BlueJ version 2.5.0 there are two server types supported: CVS and Subversion. You need to choose the correct option for the server that you are connecting to. The administrator of the repository should be able to provide this information (as well as the correct settings for the next three fields).

Server

This is the name or IP address of the server on which the repository is located. This can be a local or remote machine. Eg. cvs.myservername.org

Repository Path

This is the file system location on the server at which the repository can be found. Eg. /home/cs101/cvsroot

Protocol

This is the communication protocol to be used when communicating with the server. Which options are available will depend on the server type which has been selected.

For CVS, there are two options; *Pserver* is a CVS specific protocol that can be used for accessing a CVS repository. *Ext*, using SSH as a transport, is also supported and provides greater security, as well as being more likely to be usable through firewalls.

For Subversion, there are three options: *svn* (roughly the equivalent of CVS' pserver), *svn+ssh* (equivalent to *ext*) and *http* (a slow protocol, but which may be usable in situations where neither *svn* nor *svn+ssh* is).

The administrator of the repository will tell you which protocol to use.

If you are likely to connect to the same server in the future, you can store the server details as the default values by checking the *Remember setting for future projects* checkbox.

Once all of this information has been provided, it is possible to check whether a connection is possible with those settings by trying the *Check Connection* button. The success of creating a connection to the server using the specified username, password and server details is then given (Figure 5).

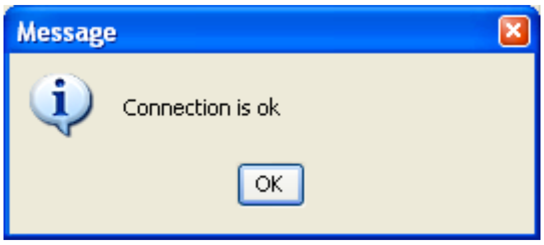

Connection is ok

Figure 5 Successful connection

If the connection was successful, you can press the Ok button to proceed with sharing the project. As this operation may involve communications with a remote machine this may take some seconds. You will see an activity bar in the bottom right hand area of the main BlueJ window. Once the operation has successfully completed, this progress bar will be removed and the three Teamwork action buttons on the left side of the BlueJ main window will now be enabled.

You project has now been copied into the repository and is available there to be checked out by other developers.

Not all of the project files are included in the repository. Some file types are excluded, the main ones being any compiled java class files found, and any generated Javadoc API documentation files for your project. These can easily be generated inside BlueJ by compiling source files and generating documentation for the project. Any other files that your project may require (such as image files) that are located within the project directory are automatically included. If your project uses a +libs subdirectory for accessing required libraries, this is also included in the repository.

5 Checking out a project

The other main way of initially interacting with BlueJ's Teamwork features is to checkout a copy of an existing shared project that is located in a shared repository. The process of checking out a project downloads a copy of that shared project from the repository on to your computer.

If you choose the *Checkout Project* menu option you will be presented with a Team Settings Dialog similar to the one described above in Sharing a project. Once the required information is given and *Ok* is pressed a *Project Selection Dialog* is presented (Figure 6) to let you select the project to checkout.

If you know the exact name of the project that you want to checkout you can specify it in the upper text area and choose *Ok*.

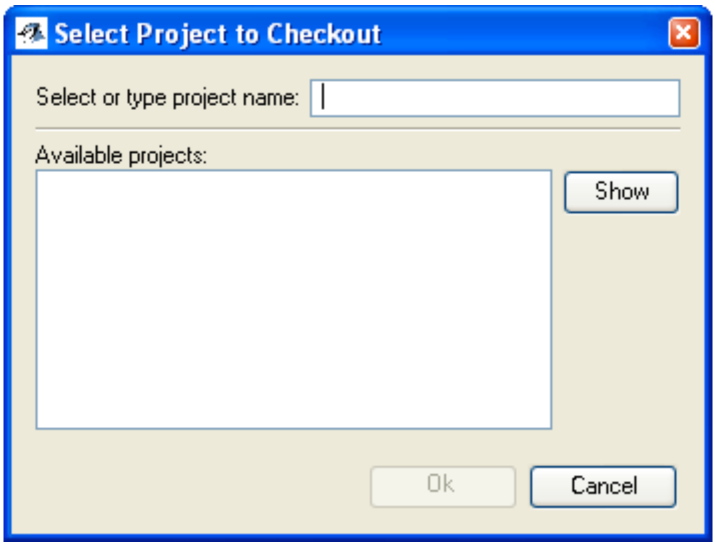


Figure 6 Project Selection dialog

Alternatively, if you are not sure of the exact project name or would like to browse the existing projects within the repository, you can select the *Show* button. This searches the repository for all available projects and displays them in a list (Figure 7). A project can then be selected for checking out. Once a project name is entered or selected the *Ok* button will become enabled.

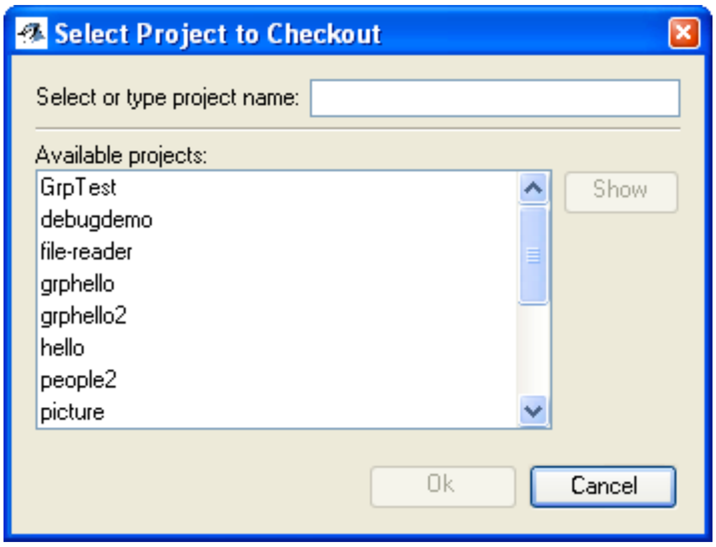


Figure 7 Browsing existing projects

The next step is to specify where on your local system you would like the project to be stored (Figure 8). All of the project files will be downloaded inside a folder that has the same name as the project. BlueJ will create this folder – you only need to specify where you would like this folder to be located.

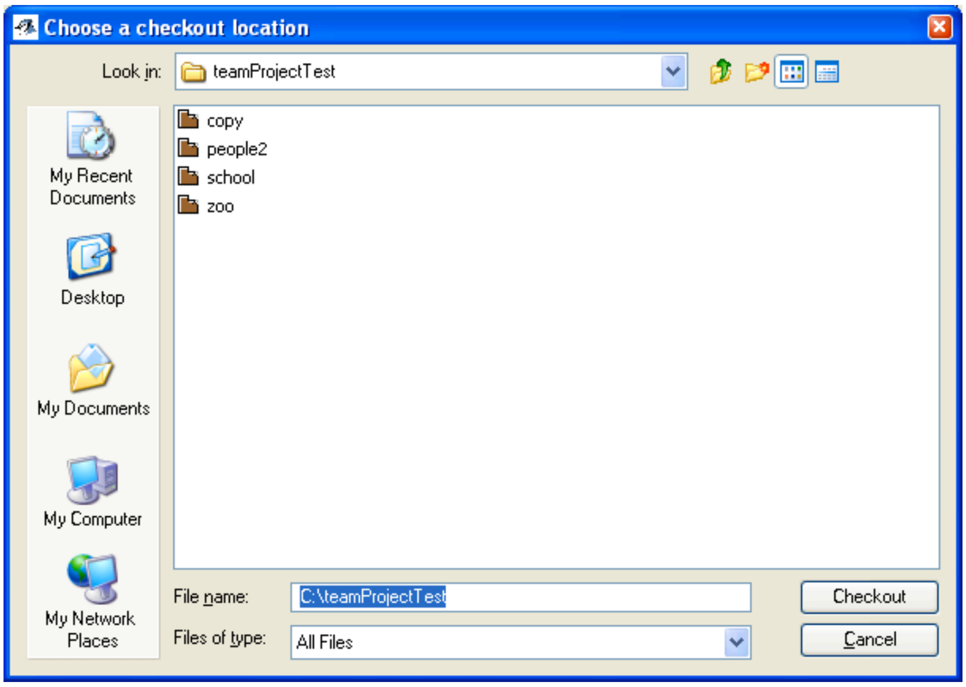


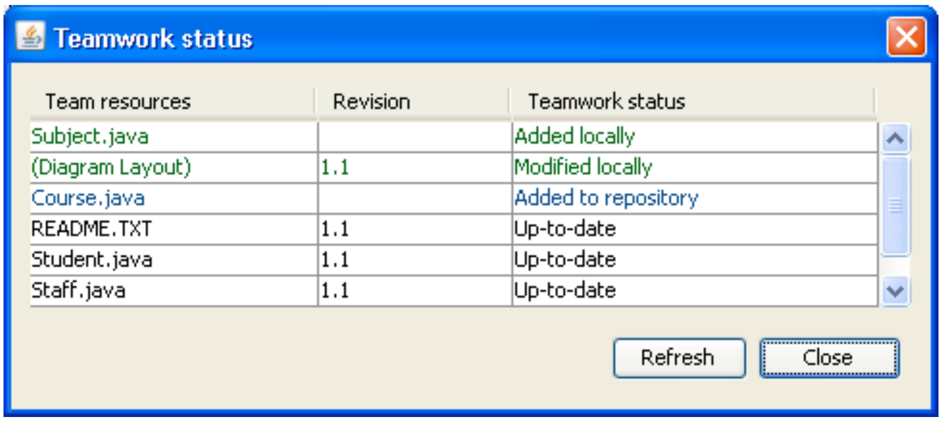
Figure 8 Choosing location for checked out project

Once you have selected the location for the checked out project and selected the *Checkout* button, the project will be copied from the repository and a local BlueJ project will be created. You can now work with this project normally and make modifications.

6 Checking the project status

A shared project exists in several separate copies. One copy resides in the repository, and one copy exists locally on the machine of each developer. The local copies are often changes by separate developers simultaneously.

The *Status* command, accessed by the *Status* button in BlueJ's main window, allows you to check the status of each class to find out whether it has been modified, either in your local version or in the repository. This information is then presented in a *Status Dialog* (Figure 9).



Teamwork status

Team resources	Revision	Teamwork status
Subject.java		Added locally
(Diagram Layout)	1.1	Modified locally
Course.java		Added to repository
README.TXT	1.1	Up-to-date
Student.java	1.1	Up-to-date
Staff.java	1.1	Up-to-date

Figure 9 Teamwork status dialog

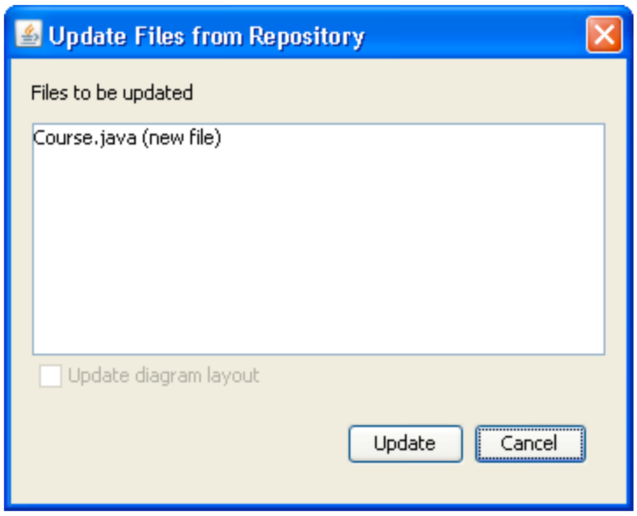
Figure 9 shows a Status dialog listing the resources for a project. The name of the resource is listed in the left column (*Team Resources*). The middle column shows the repository version number of each resource. Each time a resource is modified in the repository its version number is incremented. The status of each resource is shown in the right hand column (*Teamwork status*). In this case most of the files are listed as *Up-to-date* meaning that the repository version and your local version are exactly the same.

The "Course.java" resource is shown as having a status of *Added to repository* meaning that it is a file that has been added to the repository since your last project update, and there is no local version as yet. This file can be retrieved performing an *Update* from the repository, which is described in the next section.

"Subject.java" shows a different status: *Added locally*. As the name implies, this file has been created locally and does not yet have a repository version. You can place a copy into the repository by performing a *Commit* action, which is described in a following section.

7 Updating your local version

From time to time, as other team members make modifications to a project and commit those changes back to the repository, you will want to update your local version so that your version of the project contains those modifications. This is performed by performing an *Update*. Pressing the *Update* button in BlueJ's main window will open the Update dialog (see Figure 10), which will first check for any conflicts and inform how to resolve them. If there are no conflicts, a list of files to be updated with changes from the repository will be displayed, and you can proceed with the update (by pressing the *Update* button in the dialog).


Update Files from Repository

Files to be updated

Course.java (new file)

Update diagram layout

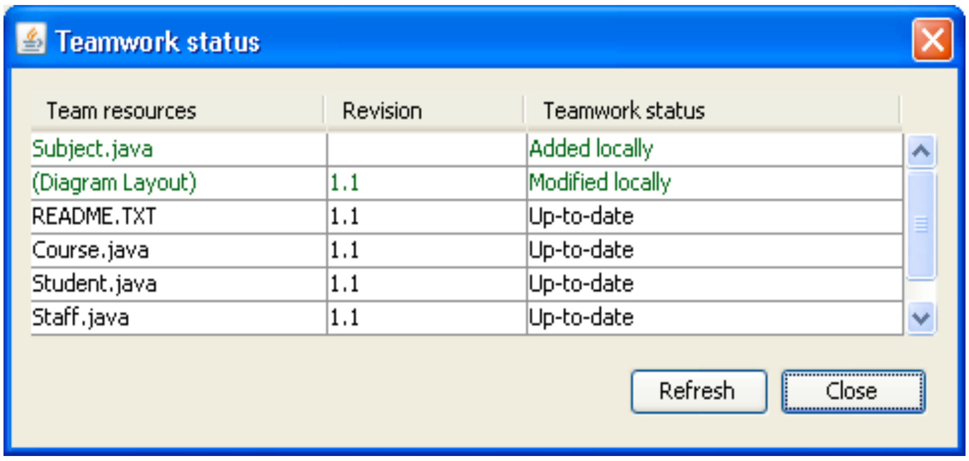
Figure 10 Update dialog

It is expected that from time to time more than one person will be modifying the same source file. It is therefore important that the version control system that enables you to work as a team does not overwrite any person's work (unless desired).

The underlying version control system will do its best to merge all modifications of a file into the final source version. From time to time, if team members are simultaneously working on the same lines of code, the version control system cannot determine what the correct version to keep is. In this situation, a *merge conflict* occurs. How to handle conflicts is described later in this document (see Resolving Conflicts). One of the easiest ways to avoid conflicts is to update your local version reasonably regularly, and to always update your project before attempting to place your modifications into the repository.

Importantly, different types of files are handled slightly differently when updating (and also when committing). Image files, data files and class libraries are not handled as text files that could be merged. If changes are made by two team members simultaneously, a conflict will be created which must be resolved by one of the team (see Resolving Conflicts).

Performing an *Update* on the project whose *Status* dialog was shown in Figure 9 will download a copy of the Course.java from the repository and place it in your project. A new class icon appears in the BlueJ main window's class diagram. Refreshing or re-opening the *Status* dialog will now show the Course.java resource as being *Up-to-date* (Figure 11).



Team resources	Revision	Teamwork status
Subject.java		Added locally
(Diagram Layout)	1.1	Modified locally
README.TXT	1.1	Up-to-date
Course.java	1.1	Up-to-date
Student.java	1.1	Up-to-date
Staff.java	1.1	Up-to-date

Figure 11 Status dialog after Update

8 Committing changes

Once you have made local changes to a project, you can choose to share those changes with others in your team by performing a *Commit* operation, which is initiated by pressing the *Commit* button in the main window. As mentioned in the *Updating your local version* section, it is advisable to perform an *Update* before committing your modifications. In any case, if you make changes to a class which is not up-to-date with the repository version, then you will not be allowed to commit that class until you have performed an update first.

It is good practice to only commit classes that compile without error, so that the project version stored in the repository is always workable. Otherwise, your incomplete work may hold up other developers.

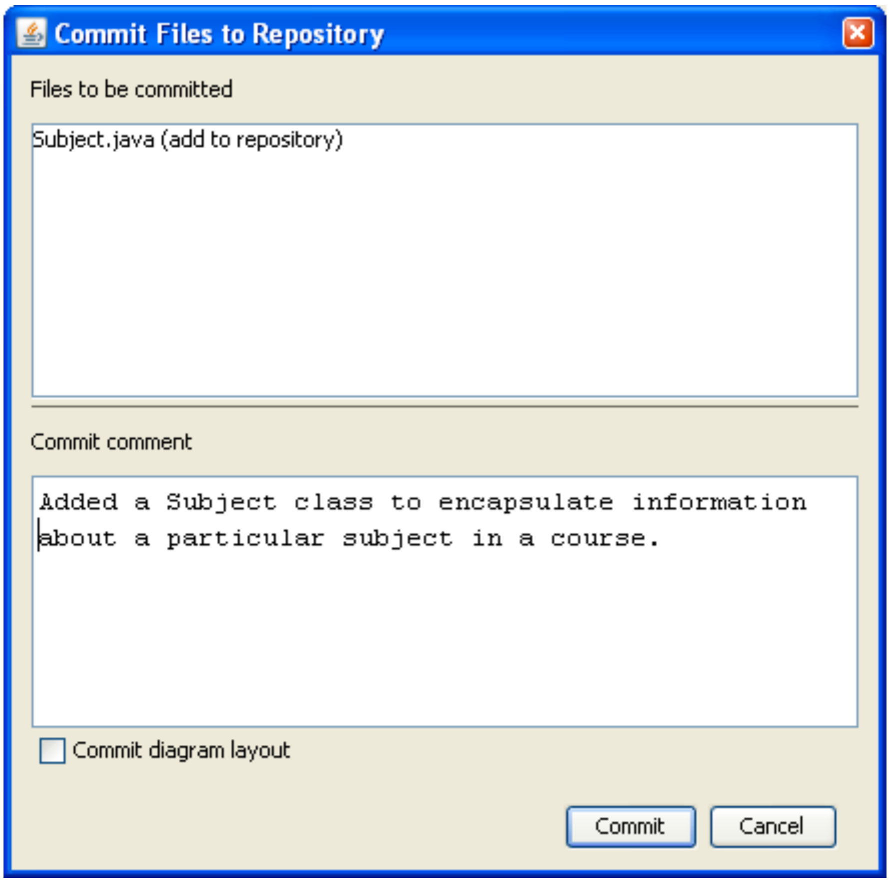
If there are no conflicts, the *Commit Comments* dialog (Figure 12) will be displayed after pressing the *Commit* button in BlueJ's main window. This dialog is used to allow you to preview the files that will be committed into the repository, and to also add some comments describing the changes you have made.

It is good programming practice to ensure that descriptive comments are added when committing changes. They provide a form of communication between developers as to the intent of changes, and an audit trail as to when, where and why certain behaviour was added (or removed).

The upper area of the Commit Comments dialog lists the files that have changed locally. These will be changed in the repository once you have pressed the *Commit* button in the dialog. Afterwards, other team members can use *Update* to retrieve your changes into their own working copy of the project.

If you have deleted files from your local version of a project, there is still a need to confirm that you want to also make that deletion in the repository. This is done also using the Commit operation. It may seem strange at first that a file that has been removed locally is included in the *Files to be committed* area, however it provides an accurate list of all of the impending changes, some of which may have been otherwise been overlooked.

If there are no changes in your local project version, the Commit button will be disabled. Below the Commit comment area there is a check box which allows the inclusion of diagram layout information in the committed files area. By default this option is unchecked and is only enabled if there are differences between your current diagram layout and the layout information stored in the repository. This feature allows developers to choose whether they wish to maintain separate diagram layouts in their local projects, or whether layout changes should also be shared.



Subject.java (add to repository)

```
Added a Subject class to encapsulate information
about a particular subject in a course.
```

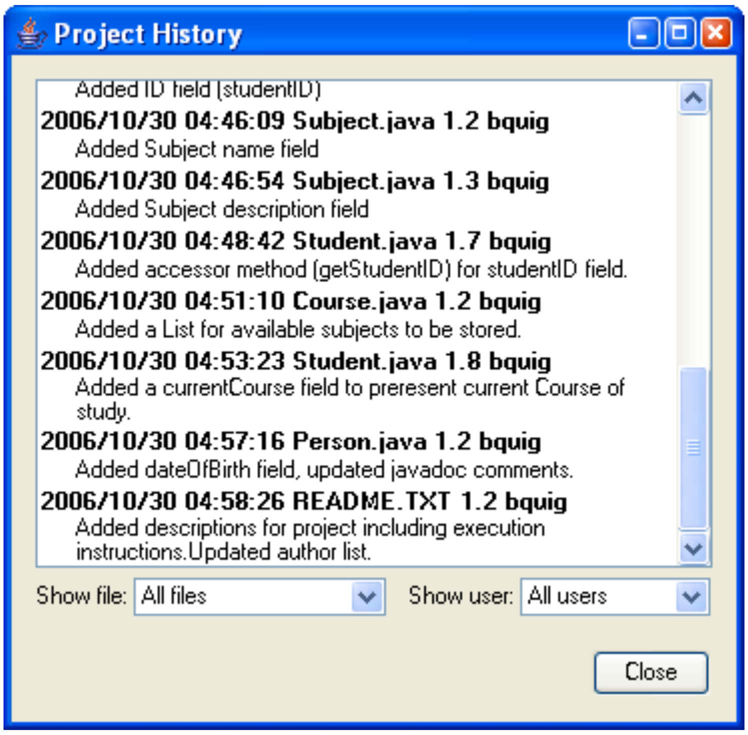
Commit diagram layout: [ ]

Figure 12 Commit comments dialog

9 Checking project history

As described above, the *Commit Comments* Dialog includes an area for the recording of comments that describe the modifications to the resources that will be committed to the repository. Writing these comments accurately and carefully is strongly recommended. This comment, along with data about which files were committed and who made the modifications, forms a *project history*. This history can be viewed at any stage (assuming you can access the repository).

Under the *Tools>Team* submenu is the menu option: *Show Project History*. This will present a dialog (Figure 13) showing all changes to the project since placed in the repository. As the amount of information for larger projects with larger teams of developers can create a lot of entries, it is possible to filter and selectively view history data by the user who has made the modification and/or the specific resource. For instance it is possible to list all changes made by *UserX*, or view all modifications to *Course.java*, or the combination of the two (all changes made by *UserX* to Course.java).



Project History

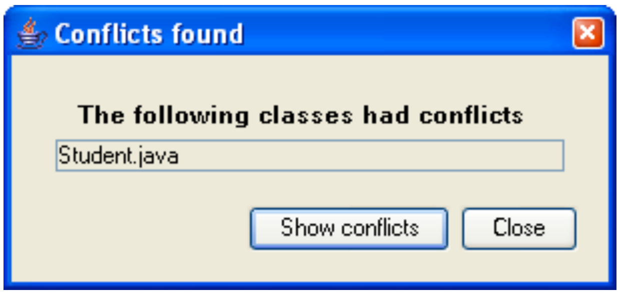
```
     Added ID field (studentID)
2006/10/30 04:46:09 Subject.java 1.2 bquig
     Added Subject name field
2006/10/30 04:46:54 Subject.java 1.3 bquig
     Added Subject description field
2006/10/30 04:48:42 Student.java 1.7 bquig
     Added accessor method (getStudentID) for studentID field.
2006/10/30 04:51:10 Course.java 1.2 bquig
     Added a List for available subjects to be stored.
2006/10/30 04:53:23 Student.java 1.8 bquig
     Added a currentCourse field to preresent current Course of
     study.
2006/10/30 04:57:16 Person.java 1.2 bquig
     Added dateOfBirth field, updated javadoc comments.
2006/10/30 04:58:26 README.TXT 1.2 bquig
     Added descriptions for project including execution
     instructions.Updated author list.
```

Show file: All files   Show user: All users

Figure 13 Project History

10 Resolving Conflicts

When updating your local project from the repository, there will be occasions where conflicts are identified between local and remote modifications. Conflicting changes in source files *(merge conflicts)* are handled by firstly alerting the user via a *Conflict Dialog* (Figure 14) and then through the mark-up of the two versions of the modified source code in the source code file (Figure 15).



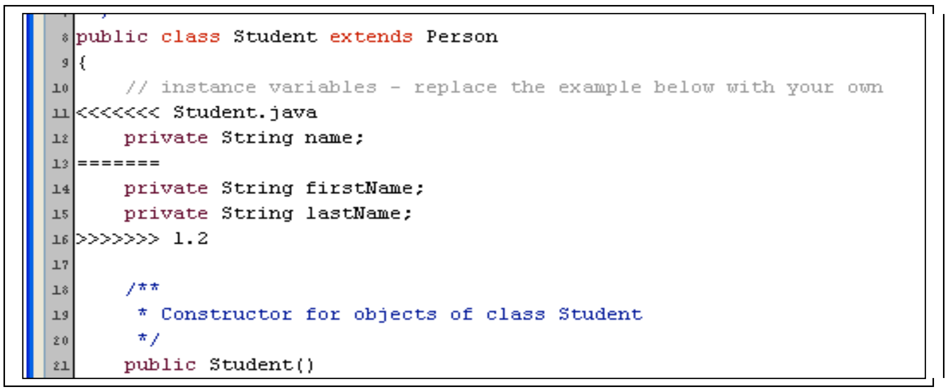
Conflicts found

The following classes had conflicts

Student.java

Figure 14 Update conflict dialog

When selecting the *Show Conflicts* button in this dialog, the source code with the conflict is shown in an editor window (Figure 15).



```
 8 public class Student extends Person
 9 {
10     // instance variables - replace the example below with your own
11 <<<<<<< Student.java
12     private String name;
13 =======
14     private String firstName;
15     private String lastName;
16 >>>>>>> 1.2
17
18     /**
19      * Constructor for objects of class Student
20      */
21     public Student()
```

Figure 15 Conflicting code marked in source code.

From lines 11 to 16 of the source code, some additional characters have been inserted to clearly mark the conflicting segment. The segment is easier to understand once the meaning of several extra characters is understood. The conflicting segment commences with line 11:

```
<<<<<<<  Student.java
```

The next lines contain the local version of the conflicting segment. The local version ends with the ======= line (line 13).

The next two lines contain the latest version from the repository. This section ends with line 16 which shows the >>>>>>> 1.2. This represents the end of the repository version (and its current version number).

To resolve the conflict you will need to choose which version (or combination of versions) of the changes you would like to keep. One other thing you will need to do is to delete the three marker lines:

```
<<<<<<< Student.java
=======
>>>>>>> 1.2
```

If the marker lines are not removed, you will get additional compilation errors, as they are not valid Java syntax. You now have a resolved merge of the file that can be used for further modifications or for committing your modifications to the repository.

There are various other types of conflict that can also occur, although they should be much less common than merge conflicts. One example is when a file has been deleted by one team member but modified by another member. Such contrary changes cannot be meaningfully merged together: Usually a conflict of this nature requires resolution by deleting the conflicting file and then performing an *Update* (in any case, the *Commit* and *Update* commands will inform you how to deal with any conflict.