Table of Contents

1 Context

Programming projects are often done in teams. This introduces the need for version control systems, to solve conflicts that may occur when multiple persons work on the same set of files simultaneously. Version control systems use so-called repositories to manage and store projects. Within the software engineering industry there is a specific need for an easy way to browse these repositories and exploit their possibilities (compare file versions, view change logs, add and remove files and so on). A Java servlet would be the ideal environment for such an application.

Two well-known version control systems are CVS[1] (Concurrent Versions System) and Subversion[2], the latter being a follow-up of the former. Particularly within LIACS [3] these systems are widely used. The most important differences:[2,10]

- In Subversion, commits are atomic. This means that an entire commit is either accepted or rejected, regardless of the number of changed files since the previous commit. In CVS, this is only guaranteed per file, which means that conflicts may occur when multiple users commit at roughly the same time.
- CVS revision numbers are per file, Subversion revisions are per commit.
- In a Subversion repository, files can be renamed while conserving version information and directories can be versioned.
- Subversion handles binary files more efficiently.

2 Problem statement

A servlet-based browser application that supports both CVS and Subversion repositories does not exist. There are examples of universal repository browsers, but they are written in PHP or other scripting languages. We particularly need the power and ease of deployment offered by Java servlets. Existing solutions written in a servlet environment are typically focused on a single type of repository (see 'Related work' section). Usually this focus is too strong to be able to integrate support for another version control system without too much effort.

Increasing the scope of an existing application to include support for a second repository type is not only a difficult task. If we want to keep the program code structural and clean, it also requires such a large refactoring that a developer would be better off starting from scratch. A new application, intended to support multiple systems in the first place, will also be much easier to extend further in the future.

3 Related work

At LIACS, an application called jCVS Servlet[4] is nowadays used to browse CVS projects. It needs jCVS[5] (a Java shell around CVS) for repository communication. Similar applications for browsing Subversion projects are, for instance, Sventon[6] and SVN Webclient[7]. These applications use the JavaSVN[8] shell to communicate with a Subversion server.

All of them, however, can only handle projects controlled by a specific versioning system. A system-independent approach to browsing and managing repositories sounds logical, but is yet unknown.

4 Solution approach

The primary aspects of the application to be build:

- It is written as a Java servlet and runs on an Apache Tomcat[9] server.
- The main features of jCVS Servlet are available.
- CVS and Subversion repositories are supported.
- It is easy to extend the application with additional functionality or system support.
- The look and feel is consistent, regardless of the repository type.

4.1 Separation is the key

The main idea behind the solution is to separate visualisation from implementation. In other words: the visual aspect should be generic and the underlying implementation should be specific for the system concerned. This can be accomplished by introducing a system-independent layer between the visual layer and the repository layer. The new layer is a model that represents a project regardless of its type. A big advantage at this point is that version control systems largely supply the same sort of information, which makes it quite easy to define a generic model.

For any kind of request to the application, the model is constructed through system-specific functions. However, in the end, it is always structurally the same. Therefore it can be presented to the user in a generic form. A key point is that the visualisation layer of the application never has to depend its behaviour on the repository type concerned. In other words: once the model is constructed, the type of project is irrelevant to the servlet.

4.2 The Module class

In programming terms, the separation into layers is accomplished by using inheritance. We define a generic 'Module' class that represents a project in a repository. The Module class is in fact the model mentioned earlier. Derived classes called CVSModule and SVNModule extend and override its behaviour when needed. This is not always the case. For example, retrieving the name of a Module is a generic function. However, retrieving the contents of a directory is different task in CVS and Subversion modules. Generic functions are only defined in the Module class. Specific functions are also defined in the Module class, but are overridden by its subclasses to correctly access the repository concerned. This way, the front-end only has to 'know' Module and its features. When Module is accessed, Java takes care of the delegation to the correct subclass, if needed.

This structure (see Figure 1) also makes it easy to add support for another system. Simply defining a third subclass of Module that overrides its behaviour when needed, would do the job.

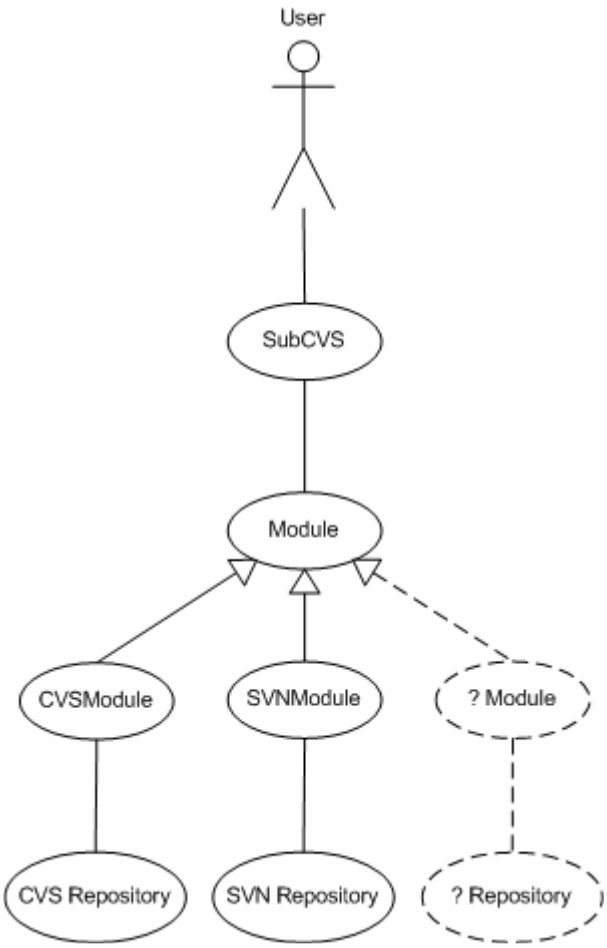

Figure 1: UML Use Case Diagram

Module and its subclasses are defined as follows. For the sake of clarity only the methods are listed.

```
public class Module {
	public Module(String name, String host, String root, String user, String pass);

	public String getType();
	public String getName();
	public String getHost();
	public String getRoot();
	public String getUser();
	public String getPass();
	public Vector getDirectoryContents(String directory, String orderBy);
	public Vector combineDirectoryContents(Vector directories, Vector files, String orderBy);
	public File getCheckoutDir();
	public File getLocalRootDir();
	public void setLocalRootDir(String checkoutDir);
	public boolean isCheckedOut();

	public void checkout();
	public String checkoutFile(String filePath, String revision);
	public void update();
	public Vector log(String filePath);
	public Vector diff(String filePath, String revision1, String revision2);
	public void add(String filePath, String logMessage, boolean binary);
	public void remove(String filePath, String logMessage);
	public void commit(String logMessage);

	public String[] combineArgs(String[] commandArgs);
	public void execute(String[] commandArgs, File dir);
	public void executeCmd(String[] args, File dir);

	public void addOutputLine(String line);
	public void addOutputStringLine(String line);
}
```

```
public class CVSModule extends Module {
        public CVSModule(String name, String host, String root, String user, String pass);

        public String getType();
        public Vector getDirectoryContents(String directory, String orderBy);

        public void checkout();
        public String checkoutFile(String filePath, String revision);
        public Vector log(String filePath);
        public Vector diff(String filePath, String revision1, String revision2);
        public void add(String filePath, String logMessage, boolean binary);
        public void remove(String filePath, String logMessage);
}

public class SVNModule extends Module {
        public SVNModule(String name, String host, String root, String user, String pass);

        public String getType();
        public Vector getDirectoryContents(String directory, String orderBy);

        public void checkout();
        public String checkoutFile(String filePath, String revision);
        public Vector log(String filePath);
        public Vector diff(String filePath, String revision1, String revision2);
        public void add(String filePath, String logMessage, boolean binary);
        public void remove(String filePath, String logMessage);
}
```

Module's constructor takes five arguments. Together they globally define a project, regardless of its type. Some specific variables are also needed. For instance, CVS and Subversion commands should be prefixed with a certain combination of username, password, et cetera. This combination is defined by the constructor of the subclass, which is called after executing the parent constructor. The combineArgs method makes sure the combination is added to each command before execution.

As you can see, CVSModule and SVNModule implement exactly the same methods. They override the corresponding methods in the parent class. This immediately shows where the CVS implementation differs from the Subversion implementation. For example, CVSModule's log method is implemented differently than SVNModule's log method. However, their results are structurally equal and can therefore be presented in a generic way.

The update and commit methods are only defined in the Module class. These are examples of generic methods. They can be used for both CVS and Subversion commands.

4.3 Directory contents

For a repository browser, a key point is to retrieve the contents of a certain directory. CVS and Subversion share the fact that information about the contents of a directory is stored in a subdirectory that is only used by the system. More specific, in a CVS repository, each directory has a subdirectory called 'CVS' in which the 'Entries' file contains all information we need. In a Subversion repository, directories have a '.svn' subdirectory in which the 'entries' file is the most important.

The getDirectoryContents methods in Module and its subclasses are used to read these files. In a CVS Entries file, each line contains a file or directory name. For files, also the latest revision, last modified date and keyword substitution mode are listed. Fields are separated by slashes ('/'). Subversion uses a more complex XML (eXtensible Markup Language) format to store entry information. Java's SAXParser class is used to parse the Subversion entries files. While implemented completely different, in the end both getDirectoryContents methods return a Vector of Entry instances. The Entry class simply extends Java's File class with the possibility to store and retrieve author and revision information.

4.4 Binary instead of library

The first attempt on developing SubCVS used the jCVS and JavaSVN libraries mentioned earlier to interact with repositories. However, they were somewhat constraining in terms of executing commands. Sometimes, the servlet needs to execute non-version control commands, for instance a 'cd' command to change the working directory. This introduced the need for Java's Runtime class and more specific, its exec method. While jCVS and JavaSVN were actually only used as a command prompt, the rest of these libraries was degraded to 'overkill'. It was far more efficient to use the ordinary CVS and Subversion command-line clients. We could then drop the libraries and execute all commands through Runtime.

This way, the output of commands was also easier to capture. Two Java threads are used to collect the output stream and the error stream. Threads are required, because both streams may produce lines in any order. This means that collecting them sequentially (first the output stream, then the error stream) may cause the application to hang. The output is both collected as a Vector of individual lines and as one big String, which in some cases is easier to parse.

Finally, Runtime's exec method takes a working directory as parameter, which makes it easy to execute a command from within a certain directory. This was a very welcome feature, because a checkout command, for instance, should be executed from within a different directory than a log command.

So, another key point is that SubCVS does not rely on third-party libraries or Java shells. It only requires the availability of the CVS and Subversion command-line clients. In other words: the locations of the CVS and Subversion executables should be included in the 'Path' environment variable. The application should function on major platforms, as long as Tomcat and the command-line clients are also supported.

4.5 Global software structure

The interaction between the different classes in the SubCVS package is illustrated in Figure 2.

- The ModuleManager class is used to store the list of modules and to retrieve the Module requested by the user. The SubCVS servlet then executes one or more commands using this Module instance.
- SVNEntryFileHandler is simply a parser for Subversion entries files.
- LogEntry represents a revision in the output of the 'log' command. The log methods in Module's subclasses return a Vector of LogEntry instances.
- Streamreaders are Threads used to collect the output of a command.
- Entry represents a file or directory in a repository.
- EntryVectorComparator compares an Entry with another Entry, for instance by name, size or revision. This way, the contents of a directory can be sorted by a specific property.

Remark: the HttpServlet, Thread, File and DefaultHandler classes and the Comparator interface are existing Java tools extended or implemented by classes in the SubCVS package.

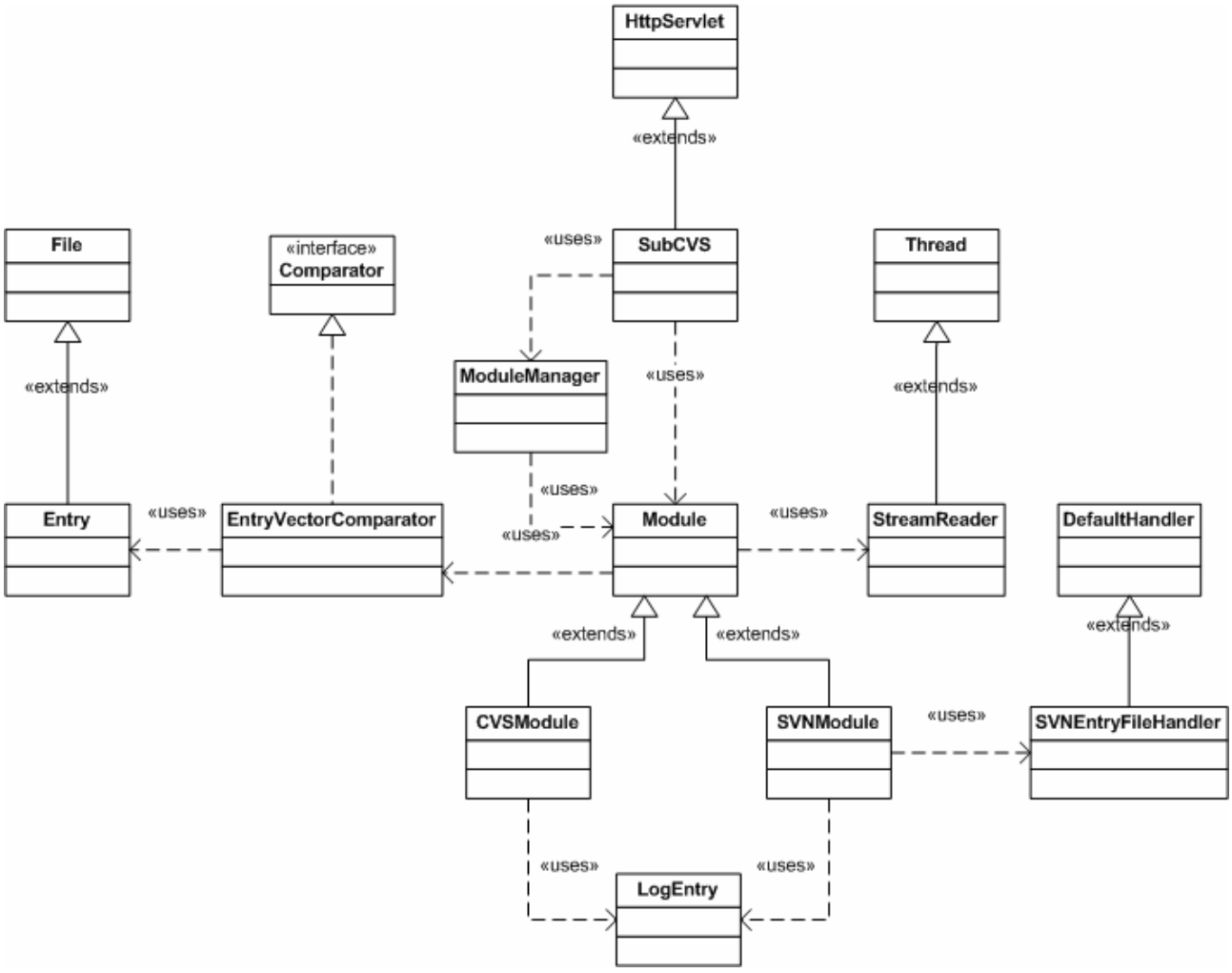

Figure 2: UML Class Diagram

5 Case study

The SubCVS application was tested on a number of different CVS and Subversion repositories within LIACS.

The user specifies all the projects he or she would like to include in a text file, called the modules file. Six fields are needed for each project:

1. Repository type
2. Project name
3. Host address
4. Path to the project directory
5. Username
6. Password

The application parses the modules file and then provides access to the modules specified. From that point, type and origin of projects are no longer issues. Users can browse and manage all projects in a generic fashion. A list of features:

- Checkout or update a local copy of specified projects.
- Browse the contents of a project. If needed, a local checkout is done first.
- View and sort files by name, size, revision, last modified date and author information. The latter is only available for Subversion projects.
- Open, add or delete files. For CVS, the '-kb' option can be specified when adding a binary file. Subversion auto-detects this.
- Explore the 'log' of a file, which contains information about all revisions of the file since it was imported into the version control system.
- View a specific revision of a file as ASCII.
- Compare two specific revisions of a file (the 'diff'). This extends the possibilities of jCVS Servlet, in which you could only compare a revision with the previous one.

All features rely on the principle of executing commands and presenting the results, just like when you are working with a CVS or Subversion client directly on the command prompt. The difference is that SubCVS provides access to common functions in a more user-friendly way. Also, users do not need to know the specifics of version control systems, because multiple systems are supported in one application. And finally, the output is presented in a more human-readable form.

The basic working cycle of SubCVS:

1. The user requests a page, for instance by clicking on a file or directory in the browser.
2. The request is translated into an appropriate command for the system concerned. In some cases a generic command is enough.
3. The command is executed and its output is captured.
4. The output is parsed and translated into a system-independent form.
5. The result is visualized on screen.

The main point during development of SubCVS was to create a framework in which this cycle could be carried out easily. Once that was accomplished, it was quite simple to add new functionality, by extending the features of 'Module' and its subclasses and providing the servlet with a way of presenting the results.

6 Conclusion

Previously, a servlet-based web application to browse repositories with support for multiple version control systems did not exist. The SubCVS project introduces this functionality. Testing the application has shown that SubCVS in fact can do this job, which makes it a very welcome addition to a software engineers set of tools.

7 Discussion

Of course in any project there is room for improvement. The SubCVS application could, for instance, be extended with the following functionality:

- change and commit files;
- add, delete and change directories;
- project history viewer;
- support for other version control systems.

8 References

1. CVS (Concurrent Versions System): http://www.nongnu.org/cvs
2. Subversion: http://subversion.tigris.org
3. LIACS (Leiden Institute of Advanced Computer Science): http://www.liacs.nl
4. jCVS Servlet: http://www.gjt.org/servlets/JCVSlet
5. jCVS: http://www.jcvs.org
6. Sventon: http://sventon.berlios.de
7. SVN Webclient: http://www.polarion.org/index.php?page=overview&project=svnwebclient
8. JavaSVN: http://www.tmate.org/svn
9. Apache Tomcat: http://tomcat.apache.org
10. David Neary, Subversion - a better CVS: http://www.linux.ie/articles/subversion
11. Karl Fogel & Moshe Bar, Open Source Development with CVS - 2nd Edition, Coriolis, 2001, ISBN 158880173X
12. Jason Brittain & Ian F. Darwin, Tomcat - The Definitive Guide, O'Reilly, 2003, ISBN 0596003188