



CHAPTER 14

Team Development with CVS

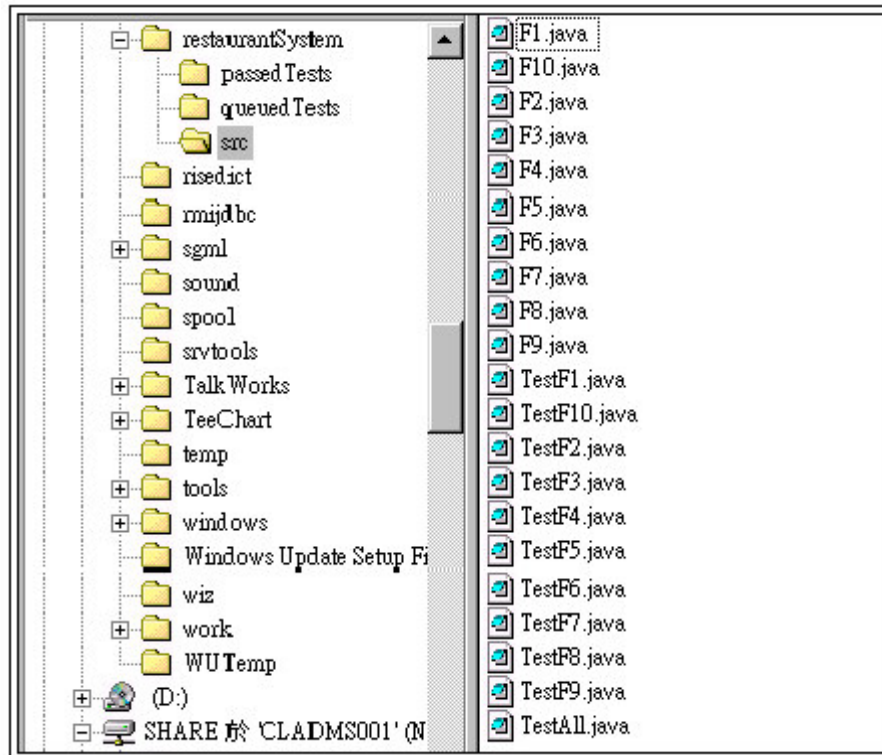


Introduction to a system

Suppose that you are developing a restaurant management system. This system consists of ten source files: F1.java, F2.java, F3.java, ..., F10.java. In addition, you have written the corresponding unit tests like TestF1.java, TestF2.java, ...TestF10.java, as well as a TestAll.java to run all the unit tests in the system. Suppose that all these .java files have been placed into the c:\restaurantSystem\src folder.

Suppose that according to the customer's requirements you have also written twenty acceptance test files such as testAddOrder.test, testAddCustomer.test and etc. At the beginning you place them into c:\restaurantSystem\queuedTests. As development makes progresses, some acceptance tests already pass. So, you move them from c:\restaurantSystem\queuedTests into c:\restaurantSystem\passedTests.

Therefore, your folders look like this:



Indeed, this way to organize files is pretty good.

Story sign up

Suppose that in order to finish the system sooner, you have hired Paul and John to develop the system with you. In the next iteration, the customer requests that user stories 3, 5, 10, 7 and 12 be implemented. You are very open-minded and hope that they can choose how many and which user stories to do. Finally, Paul chooses to do user stories 5 and 7, because user stories 5 and 7 involves a database, which is exactly Paul's strength; John chooses user story 3, because he is just a fresh university graduate and this is the simplest user story (Of course, John knows very well that in the future he has to choose more user stories, otherwise he will have no hope of any pay raise!); You choose user stories 10 and 12, two of the most difficult user stories in this iteration, because you are the best developer on the team; You must not show any hesitation. This process of the developers signing up for user stories for themselves is called "story sign up".

What are the good things about it? Look, if you insist to assign user stories 10 and 12 to John,

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

assign user story 3 to Paul and do user stories 5 and 7 yourself, will they be happy? When an unhappy developer is assigned to do something unsuitable for him, will he work efficiently?

Difficulty in team development

Now the iteration is started. You start to implement user stories 10; Paul starts to implement user story 5; John starts to implement user story 3. You have shared the restaurantSystem folder so that they can access it. Suppose that you need to modify F1.java and F2.java; Paul needs to modify F3.java; John needs to modify F4.java. Suppose that you are the quickest to get it done and have finished modifying F1.java and F2.java. So you would like to run TestAll and then run the acceptance tests in passedTests and queuedTests. However, because Paul is editing F3.java and John is editing F4.java, the system just won't compile, not to mention running TestAll and the acceptance tests.

Therefore, sharing code directly this way doesn't work. Below we introduce a way that works.

Use CVS to share code

1. You install a software package called "CVS (Concurrent Version System)" on a server. On this CVS server you create a folder designed to share source code. Such a folder is called a "repository".
2. Copy the restaurantSystem folder into the repository.
3. Everyone downloads the restaurantSystem folder from the repository onto their own computer (may put it under c:\temp to become c:\temp\restaurantSystem, or put it anywhere else). This download action is called "checkout".
4. In order to implement user story 10, suppose that you need to make the following two acceptance tests pass: testAddOrder.test and testDeleteOrder.test in c:\temp\restaurantSystem\queuedTests. To do that, you think you need to modify F1.java and F2.java in c:\temp\restaurantSystem\src. For example, you change F1.java from:

```
public class F1 {
    public int foo(int x) {
        return x+10;
    }
}
```

To:

```
public class F1 {
```

```

    public int foo(int x, int y) {
        return x+y;
    }
}

```

5. After the change, you run `testAddOrder.test` and `testDeleteOrder.test` in `c:\temp\restaurantSystem\queuedTests` and they pass. Therefore, you copy `testAddOrder.test` and `testDeleteOrder.test` from `c:\temp\restaurantSystem\queuedTests` into `c:\temp\restaurantSystem\passedTests` and then delete them.
6. Now, basically you can upload `c:\temp\restaurantSystem` back to the repository. This upload action is called "commit". However, before committing, you must run `TestAll` and all the acceptance tests in `passedTests`. Only when they all pass, can you commit.
7. When you commit, `F1.java` and `F2.java` will be uploaded. Because `F3.java` to `F10.java` have not been modified, they will not be uploaded. Similarly, `testAddOrder.test` and `testDeleteOrder.test` in `c:\temp\restaurantSystem\passedTests` will be uploaded. The `testAddOrder.test` and `testDeleteOrder.test` in `restaurantSystem\passedTests` in the repository will be deleted. In short, after you commit, the `restaurantSystem` in the repository will contain the same contents as `c:\temp\restaurantSystem`.
8. In order to implement user story 5, Paul has modified `c:\temp\restaurantSystem\F3.java` on his computer. For example, he has changed `F3.java` from:

```

public class F3 {
    public int bar() {
        F1 f1 = new F1();
        return f1.foo(10);
    }
}

```

To:

```

public class F3 {
    public int bar() {
        F1 f1 = new F1();
        return f1.foo(20);
    }
}

```

9. Finally Paul succeeds in making the acceptance tests for user story 5 pass. Therefore, he is about to commit. If CVS allowed him to commit, his `F3.java` would be uploaded and he would not notice any problem. However, the `F1.java` in the repository would not work with his `F3.java` (a compile error would occur):

```

public class F1 {
    public int foo(int x, int y) {
        return x+y;
    }
}
public class F3 {

```

```
public int bar() {
    F1 f1 = new F1();
    return f1.foo(20); //COMPILE ERROR!!!
}
}
```

Fortunately, CVS will not let him commit. When Paul tries to commit the whole `c:\temp\restaurantSystem` folder, CVS will find that Paul's `F1.java` is an older version of the one in the repository. Therefore, it will refuse to commit. Now what should Paul do? Before Paul commits, he should check if the repository contain any updated files. If yes, he should download them. This action of downloading the updated files is called "update". It is different from checkout. checkout performs the first download. Every download after that is an (incremental) update.

10. So Paul performs an update and downloads the updated versions of `F1.java` and `F2.java`. At the same time his `passedTests` folder will get two new files: `testAddOrder.test` and `testDeleteOrder.test`. Because `F1.java` and `F2.java` have been updated, he needs to compile again and run `TestAll` and `passedTests` again. However, currently the system simply won't compile. He has to fix the compile error first. How to do that? He can do it any way he wants, including reverting `F1.java` back to the original version. However, the precondition is that he must ensure that all the tests in `passedTests` continue to pass (because you committed first, `passedTests` now contains `testAddOrder.test` and `testDeleteOrder.test`).
11. Suppose that after Paul's hard work, he finally fixes all compile errors, makes `TestAll` and `passedTests` pass. He can try to commit. If before him John has committed, once again he will be unable to commit. He needs to do an update, fix all errors, make `TestAll` and `passedTests` pass and then try to commit again. From this we can see that we should commit as soon as possible. The person who commits late may have to clean up all the mess. In fact, even before Paul succeeds in making the acceptance tests for user story 5 pass, as long as `TestAll` and `passedTests` pass, he can commit to reduce the chances of having to clean up the mess. This way of frequently integrating code is called "continuous integration".

Different people have changed the same file

Suppose that in order to implement user story 5, Paul needs to modify not only `F3.java`, but also `F1.java`. For example, he changes `F1.java` from:

```
public class F1 {
    public int foo(int x) {
        return x+10;
    }
}
```

To:

```
public class F1 {
    public int foo(int x) {
        return x+10;
    }
    public int bar(int x) {
        return x;
    }
}
```

When Paul updates, the F1.java in the repository is your version:

```
public class F1 {
    public int foo(int x, int y) {
        return x+y;
    }
}
```

If Paul hadn't changed F1.java, CVS would use your version of F1.java in the repository to overwrite his copy of F1.java (because your version is an update based on his version). Or, if Paul did change F1.java but the F1.java in the repository was still the old version, CVS would simply do nothing (because his version is an update based on the version in the repository). However, now CVS finds that Paul's F1.java is no longer the original F1.java and that the F1.java in the repository has also been updated. None of these two versions is an update based on the other. They are two different updates based on the original version. Therefore, CVS will merge the copy in the repository into Paul's local copy, making Paul's version the most updated version based on these two versions:

```
public class F1 {
    public int foo(int x, int y) {
        return x+y;
    }
    public int bar(int x) {
        return x;
    }
}
```

Of course, after that Paul needs to re-compile, make sure TestAll and passedTests continue to pass and then try to commit.

There are some cases in which CVS cannot merge two versions smoothly. For example, as supposed before, you have changed F1.java to:

```
public class F1 {
    public int foo(int x, int y) {
        return x+y;
    }
}
```

But Paul has changed F1.java to:

```
public class F1 {
    public int foo(int x, int y) {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agilekills.org

```
        return x+2*y;
    }
}
```

When CVS tries to merge, it finds that you and Paul have changed the same part of F1.java and therefore can't decide which version it should keep. In this case, the merged F1.java will be like:

```
public class F1 {
    public int foo(int x, int y) {
>>>>>>>>
        return x+2*y;
<<<<<<<<<
        return x+y;
    }
}
```

Markers like "<<<<<<<<" make F1.java fail to compile, advising Paul that he must manually check the code and decide which version to keep. When CVS cannot merge as shown in this case, we say there is a "conflict".

Add or remove files

If you add a new file like F11.java in c:\temp\restaurantSystem\src, when you commit, CVS will not automatically upload it. As a result, the system in the repository probably will not compile (because F11.java is missing). To make it include F11.java, you need to "add F11.java to CVS".

Similarly, if you delete F1.java from c:\temp\restaurantSystem\src, when you commit, CVS will not automatically delete the F1.java in the repository. To make it delete F1.java, you need to "remove F1.java from CVS".

It is very easy to forget to add files to or remove files from CVS. Stay alerted!

Different people have added the same file

Suppose that you have added F11.java and Paul has also added a file with the same name (F11.java). Suppose that you commit first. When Paul updates, CVS will check the relationship between Paul's F11.java and the F11.java in the repository. CVS will find that neither one is an update based on other, nor they are two different updates based on the same version. Instead, they are two completely independent files. In this case, CVS will not merge them. It will simply treat it as an error and refuse to update. To solve this problem, Paul may remove his F11.java from CVS, rename his F11.java (e.g., to F11Old.java), update again to get the F11.java from

the repository, manually merge F11Old.java into F11.java and then delete F11Old.java.

Collective code ownership

As mentioned above, you, Paul and John have the right to modify any files (F1-F11.java) in the system without getting approval from anyone as long as TestAll and passedTests continue to pass. Therefore, you, Paul and John collectively own all the code in the system. This is called "collective code ownership".

How to use command line CVS

There are various CVS clients. Some have a command line interface and some have a GUI (e.g., WinCVS). Here we will introduce how to use the command line CVS client in a Linux/Unix environment. Suppose that a repository is in the /var/cvs folder on a server with a DNS hostname c001.cpttm.local.

We need to perform the configurations below:

- Ensure that the developers can use ssh to login to c001.cpttm.local.
- Ensure that the developers have executed ssh-agent, input their passwords and are able to login to c001.cpttm.local repeatedly without inputting a password again. For the details, please see the man page of ssh-agent.
- CVS are installed on the computers used by the developers.
- An environment variable named "CVS_RSH" has been created and set to the value of "ssh" on those computers.
- An environment variable named "CVSROOT" has been created and set to the value of ":ext:c001.cpttm.local:/var/cvs" on those computers.

The commonly used commands are shown below:

checkout	cd /temp cvs checkout restaurantSystem
----------	---

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

update	cd /temp cvs update restaurantSystem
commit	cd /temp cvs commit restaurantSystem
Add F11.java	cd /temp/restaurantSystem/src cvs add F11.java
Remove F11.java	cd /temp/restaurantSystem/src cvs remove F11.java

References

- Official web site of CVS: <http://www.cvshome.org>.
- <http://www.c2.com/cgi/wiki?CvsTutorial>.
- <http://www.c2.com/cgi/wiki?ContinuousIntegration>.
- <http://www.c2.com/cgi/wiki?CollectiveCodeOwnership>.
- Ron Jeffries, Ann Anderson, Chet Hendrickson, Extreme Programming Installed, Addison-Wesley, 2000.

Chapter exercises

Problems

1. Suppose that you have checked out the system. In each of the following cases, what will CVS do?
 - You modify F1.java and then update.
 - You commit immediately.
 - You modify F1.java and then commit. However, Paul has committed first. He has modified F2.java.
 - You modify F1.java and then commit. However, Paul has committed first. He has modified F1.java.
 - You delete F1.java in Explorer and then commit.
 - You create a new file F12.java and then commit.

Sample solutions

1. Suppose that you have checked out the system. In each of the following cases, what will CVS do?

- You modify F1.java and then update.

CVS will do nothing.

- You commit immediately.

CVS will do nothing.

- You modify F1.java and then commit. However, Paul has committed first. He has modified F2.java.

CVS will refuse to commit and ask you to update first.

- You modify F1.java and then commit. However, Paul has committed first. He has modified F1.java.

CVS will refuse to commit and ask you to update first. When you update, if you and Paul has modified different parts of F1.java, CVS will merge the changes. If you two have modified the same part of F1.java, CVS will tell you that there is a conflict. In that case you will have to clean up the markers and merge the changes manually.

- You delete F1.java in Explorer and then commit.

CVS will do nothing. You should have removed it from CVS first.

- You create a new file F12.java and then commit.

CVS will do nothing. You should have added it to CVS first.

