REQUIRED CS 314 Eclipse and Git Tutorials

Introduction

We will be using the Git source control system in CS 314. Git is a source control system that is frequently used for collaborative development, and GitHub.com is one of several hosting sites for both public and private Git repositories that can be accessed over the internet. Dr. Bieman has obtained a free private repository area for student teams to use this semester. We have created a private Git repository for each CS 314 team. These repositories were created with an initial version of the Java Adventure Cave Game, as an Eclipse project, along with an initial class diagram of the system, and a sample object diagram of a cave.

Git can be used in multiple ways. We will be using the integrated plugin (EGit) that comes with Eclipse releases since the Kepler release. We will also be using the collaboration tools that are available from the GitHub.com website. You will use the Issue tracker associated with each student group repository. You will use the Pull Request Code Review System for discussions among your team members. You will therefore need to regularly login to GitHub.com and access the issue tracker to see issues and discuss their resolution with your teammates, and the Pull Request Code Review System to discuss changes to project artifacts with your teammates.

We will be using a particular workflow with Git, and this will help you gain experience with daily tasks you will encounter during your career, such as creating and managing branches, merging changes, and developing and managing unit tests.

Some information on using the Pull request code review system can be found in a posting by Scott Chacon. This posting can be found at: http://scottchacon.com, then click on the GitHubFlow link dated Aug 31, 2011.

An excellent Eclipse-Git tutorial that covers many of the details described below to set up Eclipse-Git can be found at: http://www.vogella.com/tutorials/EclipseGit/article.html

Finally, Eclipse-Git user documentation can be found at: http://help.eclipse.org/mars/index.jsp
Expand the EGit Documentation section, and go to 'EGit User Guide'.

Work through the following tutorials that are in this document:

GitHub Commands you will be using:

Pull: Fetches changes and merges them in one step – ONLY use the *Pull Request Code Review System* on GitHub.com to initiate this command so that your teammates can review your changes.

Eclipse Git Commands you will be using:

Branch: Creates a new local branch or switches to an existing local branch.

Right click on project name, **Team->Switch To** or branch icon in the tool bar 

Fetch: Gets all the data on the remote branch that you don't have in your local version of the branch. It does NOT merge any changes for you.

Right click on project name, **Team->Fetch from Upstream** or fetch icon in the tool bar 

Push: Sends all the data in your local branch to the remote branch.

Right click on project name, **Team->Push to Upstream** or push icon in the tool bar 

Merge: Incorporates changes from another branch into the current branch. Most of the time this is straightforward. If there are conflicts there is a merge tool in Eclipse to help resolve conflicts.

Right click on project name, **Team->Merge** or merge icon in the tool bar 

History: Show all the commits and changes that have gone into a branch. The History tab lists the commits and a simple graphical view of the branch. If other branches were created from it or merged into it this is shown in the view. If you click on any commit number, the files changed and the commit messages are shown.

Right click on project name, **Team->Show in History**

REQUIRED Eclipse-Git Workflow

Rules:

You must follow these rules when working with your Git repository.

1. You MUST create individual branches for all work that you do. Each person needs to have their own branch(es) for anything they are doing. It is better to create a branch that is named for the work that you will be doing on it, such as fixing compiler warnings, adding unit tests, adding functionality, etc. You should divide the work between the people on your team, so that each person has about the same amount of work to do.

   This rule is common in industry – the master branch is usually considered deployable; it builds, runs, and doesn't have test issues. In order to support this, all developers must work on other branches.

2. You MUST check in small changes; don't wait until an entire task is complete before checking in your changes.

   This rule is also common in industry – often there is a single problem with a check in that interferes with something else and breaks the system, so the change has to be backed out. If you check in often, then less will have to be backed out in the event of a problem.

3. You MUST include descriptive check in comments in every check in. Comments for code changes made from A2-A5 must include the user story and task for which they were made.

   This rule is related to the previous one; if there is a problem, descriptive comments make it much easier to find the problem changes and back them out. These sorts of problems are often found during integration or beta testing, and it can be really difficult to figure out what the change was that caused the problem. If it can be found as a particular check in then it usually can be fixed fairly easily. This is also a necessary rule for collaborative development where developers are not always working together. You will have this challenge in CS 314 simply because you and your teammates will most likely not be able to always work together, and it is critical that you all know what is going on.

Guidelines:

You will find that working with branches is much easier if you follow these guidelines.

1. Push your work to the GitHub repository often.

Since you will be using the Eclipse plugin, there is a button that automatically does this when you commit a change to your local copy of the repository. Doing this makes it easier for your teammates to get your latest changes and merge them into their branches. Similarly, it makes it easier for you to get their changes and ensure there are no problems.

2. Keep your local copies of your teammates' branches up to date by fetching their branches to your local repository often.

   This is an Eclipse command so it is also pretty easy to perform. You can specify which branches to update from the GitHub.com repository.

3. Merge your teammates work into the branch you are working on regularly.

   This can be done using Eclipse as well. Merging with Git is usually straightforward and there aren't conflicts unless you are working on the same file in the same place as one of your teammates. Keeping your branch merged with your teammates' work helps avoid this problem, and partitioning the work among your team can also help. There are also tools that help you figure out what happened when conflicts occur.

Tutorial 1. Setting up Eclipse for Git (EGit)

There is a very good tutorial that covers more of the capabilities of EGit at Vogella.com regarding Eclipse and Git. It can be found at:
http://www.vogella.com/tutorials/EclipseGit/article.html

There is also good documentation from the Eclipse site: http://help.eclipse.org/mars/index.jsp. Expand the EGit Documentation section, and go to 'EGit User Guide'.

Tutorial 1 will help you set up the EGit plugin for your local Eclipse project.

1. Set the location for your repositories on your local system. The default location can be changed from Window->Preferences->Team->Git, and is called 'Default repository folder'. It is often in your home directory, and called 'git'. This is fine.
2. Go to Window->Preferences->Team->Git->Configuration. The table lists entries in a '.gitconfig' file, also usually in your home directory. If the table does not have a bolded key called 'user', you need to add this with the Add Entry button. The add form requires a 2-part key and a value. You need keys for your email and real name. For the first key, use 'user.email', then add your email as the value. For the second key, use 'user.name' and enter your name. These values will be used at GitHub.com during communication with the repository, so the email must match the email you used to create your GitHub account. You also need to configure Eclipse so that branch tracking with the GitHub repository works correctly. The key is 'branch.autosetuprebase' and its value needs to be 'always'.

   There are other items you add to the configuration. The Vogella tutorial lists some useful items, for example the key 'push.default' with value 'simple'.

3. You can add Git commands to the toolbars as follows. Got to Window->Perspective->Customize Perspective. If possible, check the Git box on the 'Tool Bar Visibility' tab. If this is grayed out, then go to the 'Action Set Availability' tab, check the 'Git' box. This will show the actions in the in the Menubar details frame and the Toolbar details frame. Select the OK button. Next check the 'Git Navigation Actions' box and select the OK button. You can then go back to the 'Tool Bar Visibility' tab and check the Git box.
4. You can specify which files Git should ignore in a file called .gitignore which is usually in your home directory. The Vogella tutorial lists some of these files, for example binary files. We have added a .gitignore file to your team repository.

Tutorial 2. Connecting Eclipse to Your GitHub Repository

Now you are ready to create a local copy of your GitHub repository that will be used by Eclipse.

1. First show the 'Git Repositories' view by going to Window->Show View->Other. In the window that comes up, click the expand icon by 'Git' and select 'Git Repositories'.
2. There are several icons that will be displayed in this view. Mouse over them to find the one labeled 'Clone a Git Repository and add the clone to this view' and select it, or click the link labeled 'Clone a Git repository'.

3a. You need to enter the URL of your repository on GitHub.com. To find this, login to GitHub.com and navigate to your CS314 repository. There will be a text field with the URL to this repository, prefaced with an https. Copy this URL to the Eclipse 'Clone Git Repository' form. The Host and Repository path fields will be automatically filled in. You can add your GitHub user name and password to be stored locally if you want, or you can leave these fields blank. Click the Next button.

3b. The next window allows you to pick the branches that will be cloned and copied to your local system. The only branch that should exist is 'master', and it will be selected by default. Press the Next button.

3c. The next window lists the directory where the clone will be stored, and it is in the default directory that you added to the configuration file, for example your <home directory>/git.

IMPORTANT: Check the box under the heading 'Projects' that is labeled: 'Import all existing Eclipse projects after clone finishes'. This will create your Eclipse project from the cloned repository. Click the Finish button and the clone and project creation will begin. If you have added your GitHub credentials to the Eclipse Safe Store then the operation will complete without further work from you. If you have not added the credentials, you will be prompted at least once for this information.

Tutorial 3. Branching

Your task is to correct compiler warnings and add comments in the imported project, and to propagate these changes into your repository on GitHub.com. This work will be done on multiple branches. In this example, one person (P1) will be fixing compiler warnings, another (P2) will be adding required project comments to all code files, and the third (P3) will be adding additional useful comments to the code. Each person will be working on a different branch so that the changes can be incrementally checked in and merged. Specific changes are discussed in each of the 'Branching' sections below.

Branching – P1 creates a branch to fix compilation errors

1. The first thing you need to do is make sure your local copy of the master branch is up to date with the GitHub repository. To do this, highlight the project name (which should list 'master' in square brackets next to the name) and click the 'Fetch' icon on the tool bar or right click the project name, go to 'Team' and choose 'Fetch from upstream'. This will fetch all updates from the GitHub repository for the master branch.
2. To create a branch, highlight the project name and click the 'Branch' icon on the tool bar or right click on the project name, go to 'Team' and then 'Switch to', and choose 'New Branch'. A new window will be displayed. You can get to this same window by clicking on the branch icon in the tool bar, and clicking the 'New Branch' button.

   The area labeled 'Source:' has a branching icon. Click the 'Select' button on the right of this label. A new Select Source window will appear. Expand the 'Remote Tracking' listing if it is not expanded, and select 'origin/master' as the source and click the 'OK' button. The new branch will be to fix the compiler warnings, so in the 'Branch name' field enter your initials and the name 'fixWarnings'. Leave the other defaults and click the 'Finish' button. The new branch is created, and the project is changed to that

branch. This is shown via the branch name in the square brackets after the project name in the Package Explorer view.

3. Push the new branch up to the GitHub repository. Do this by clicking the 'Push' icon on the tool bar or right clicking on the project/branch name, go to 'Team' and choose 'Push to Upstream'. This will create the branch in the GitHub repository. A window will appear that allows you to add more comments if you want, and to OK the push. Click OK.
4. Configure the fetch command so that it fetches changes from different branches in the Git repository. To do this, click the 'Fetch' icon on the tool bar or right click on the project name, and go to Team->Fetch from upstream. Press the 'Configure' button quickly, before it defaults to the master branch. A new window will be shown, titled 'Configure Fetch'.

   There is a single default line in the 'Ref mappings' section, listing origin/master or origin/*. Highlight this line and delete it. Next click the 'Advanced' button. This will bring up another window. Use the dropdown menu for the Source reference and select the new branch you just created ('<your initials>fixWarnings'). The Destination reference will list the remote branch of the same name. Click the Add Spec button. Click the Finish button. You can now click on the 'Ref mappings' entry and the 'Dry Run' button to see if there are any commits on the GitHub repository for that branch which you do not have locally. Click the 'Save' button to exit the configuration.
5. Fix the warnings. Use the 'Problems' tab to see them – they exist in 2 files. Fix one file and check in those changes and then fix the other file and check in its changes since the warnings in the 2 files are not related. Fix the first file and save the changes. Close the file in the editor. Make sure the 'Git Staging' tab is shown in your Eclipse window. It has several parts: 'Unstaged Changes', 'Staged Changes', 'Commit Message', 'Author', 'Committer', and 2 buttons, labeled 'Commit and Push' and 'Commit'. After your changes to the first file, the file name should listed in the 'Unstaged Changes' area. Drag this name and drop it into the 'Staged Changes' area. Type a descriptive message into the 'Commit Message' area. Pop up warnings will occur if you type in more than 1 line, but make it as descriptive as you can – what the change was and why you did it. If there are several files that have been changed, this message should describe why they were changed. Finally, click the 'Commit & Push' button. This will commit your changes to the local copy of your branch and push them to the remote.
6. Look at the GitHub repository. You should see that there is a section that lists your recently pushed branch and it shows the '<your initials>fixWarnings' branch. If you click on this name, the GitHub repository changes to that branch. If you click the 'Compare & pull request' button associated with this branch, you will see a new window that is titled 'Open a pull request'. This is the Pull Request Code Review System we will be using for collaboration. For now, just scroll down the page, and you will see the commits that have occurred for this branch, and further down there is a code listing that shows changes to the files that were made as part of the commits.

   Just above the code listings there are 2 buttons, 'Unified', and 'Split'. If you click the 'Unified' button you see the code changes within a single listing window, and if you click the 'Split' button you see file changes side-by-side. DO NOT open a pull request – instead scroll back near the top of the window and click the tab labeled '<> Code' to take you back to the main repository page.
7. Fix the warnings in the second file, then proceed as in step 5 to stage and commit and push the changes to your branch.
8. Update the overview.txt file with your contributions and you statement of compliance to the CS honor code, then proceed as in step 5 to stage, commit, and push the changes to your branch.

Branching – P2 creates a branch to add required project comments to all files

Repeat steps 1 – 6 above, but use the branch name '<your initials>addProjComments'. Each source file should begin with comments indicating the name of the file, a brief description of the purpose of the file,

author(s) and date of creation. Also include the date and author of each update, along with a brief description of modifications. Some files have some of these comments from Dr. Bieman's original work, but others do not. In addition, some files describe compiling and running using the command line, and these comments have the wrong package name. Fix these comments at the beginning of the file, add an update section where you briefly describe what you have done to the comments.

You need to make these changes with at least 2 commits. Remember, the purpose of small commits is that if there is a problem you don't want to have to back out lots of changes – just the ones that really caused the problem.

Branching – P3 creates a branch to add functionality comments to the code

Repeat steps 1-6 above, but use the branch name '<your initials>addFuncComments'. There are lots of places you can better document this code. Do not comment trivial code or make trivial comments. One approach is to lay the groundwork for using Javadoc. Be sure to add an update section to the comments at the beginning of the file.

You need to make these changes with at least 2 commits. Remember, the purpose of small commits is that if there is a problem you don't want to have to back out lots of changes – just the ones that really caused the problem.

Tutorial 4. Merging Part 1: Merging the remote master to your local working branch

The CS 314 development process requires that the master branch always is deployable. So you need to add your changes to it in 2 parts. The first part is to make sure that any changes that have been already added to the master work with your proposed changes. This is covered in tutorial 4. The second part is to request a code review from your teammates, and when any issues have been resolved, to merge your changes into the master branch. This second part is covered in tutorial 5.

The Egit help documentation has screen shots of the various operations described below. It can be found at: http://help.eclipse.org/mars/index.jsp

Expand the EGit Documentation section, go to 'EGit User Guide', then 'Tasks', then 'Merging', and then 'Merging a branch or a tag into the current branch'

You now need to merge any changes made on the remote master into your local branch. Then you need to push these changes from your local branch to the GitHub repository, and then merge them into the master on the remote. Do this as follows.

1. First close any files that are open in the editor. Then switch branches in Eclipse to the master branch. Do this by right clicking on the project name and choosing the Team->Switch To command, and choose the 'master' branch. Alternatively, you can click the branch icon in the toolbar, which will open a window titled 'Branches'. Expand the 'Local' section and click on the 'master' branch. Next click the 'Checkout' button, which just actually switches you to the master branch in your local repository.
2. Merge any changes from the GitHub repository to this local copy of the master. To do this, click the 'Merge' icon on the tool bar or right click the project name and choose Team->Merge. In the window titled 'Merge master', the local master should be checked, and you need to click the 'Remote Tracking' section to expand it, then highlight 'origin/master'. Make sure the 'Commit' radio button is selected and then click the 'Merge' button.
3. Switch branches to the branch where you have made changes. We will assume you are P1 and the branch is '<your initials>fixWarnings'.

4. Merge any changes from the GitHub repository to this local copy of the branch, just like you did for the master branch. If you are sure that you are the only person who is working on this branch and if you have pushed all of your local commits to the GitHub repository you can skip this step.
5. Merge the local master branch into your local branch (‘<your initials>fixWarnings’) branch. The merge will be using your local branches. Click the ‘Merge’ icon on the tool bar, or right click on the project name (which should indicate that the current branch is the ‘<your initials>fixWarnings’ branch), and choose the Team->Merge command. A new window will appear.

   The local branch you are viewing is marked with a check mark. Click on the name of the local master branch – make sure to click on the local copy of this branch. Select the ‘Commit’ radio button to commit to the merge. Click the Merge button.

   There shouldn’t be any conflicts, but if there are, you must resolve them. The EGit user documentation can help (‘EGit User Guide’, then ‘Tasks’, then ‘Merging’, and then ‘Resolving a merge conflict’, and the Vogella tutorial has a section on resolving conflicts (section 21.2). EGit has a Merge Tool to help resolve conflicts. If there are conflicts, you will be making changes on your local ‘<your initials>fixWarnings’ branch to resolve them. Make sure your changed branch compiles and runs. Commit any changes and push them up to the GitHub repository. Then try the merge again, and this time it should work.
6. Make sure the program compiles and runs. Make any necessary changes to accomplish this on your local ‘<your initials>fixWarnings’ branch and push the changes to the remote.
7. If you had to fix merge conflicts in step 5 or make changes in step 6, start again at step 1.

Tutorial 5. Merging Part 2: Request reviews of your changes with the Pull Request Code Review System

1. Now you need to request that your teammates review your changes using the GitHub.com Pull Request Code Review System. To do this, go to the GitHub repository. Click the dropdown labeled ‘Branch’, and choose your branch name. Then click the ‘New pull request’ button next to the branch name.
2. The ‘Open a pull request’ form is displayed. You may have to change the branch names that are shown as defaults. To do this, use the dropdowns to make sure that the base is the master branch and the compare branch is your branch. Scroll down the page to see the commits that have been made to the branches and the changes to the files. Scroll back up to the ‘Add comments’ section and type in a message to your teammates that explains the reason for the changes that you made and briefly what they were, and what files you changed. If there is an issue tracker item associated with these changes, reference it in the reason why you made the changes. Click the ‘Create Pull Request’ button to send the message to your teammates.
3. Your teammates should see a new pull request when they are logged into the GitHub repository. There is a tab at the top of the window labeled ‘Pull requests’ with a number next to it. If you click on that tab, all open pull requests are visible. Click on one of them to open the window about the request. There is a title with 3 tabs just below this title, ‘Conversation’, ‘Commits’, and ‘Files changed’. Everyone on the team needs to review the changes you are proposing, by clicking the ‘Files changed’ tab and seeing what you want to merge into the master branch. Again there is a ‘Unified’ and ‘Split’ set of buttons to view the changes side by side or sequentially. In this case only the changed lines are shown by default, but there is a ‘View’ button that shows the entire file. Your teammates need to review your code changes, then go back to the ‘Conversation’ tab and enter their comments, adding them to the conversation by clicking the ‘Comment’ button. DO NOT click any other button on the page.

4. You need to monitor the conversation and if anyone has suggestions or issues with your proposed changes you need to fix them. You can suggest changes in the conversation to make sure you have a fix before proceeding. Only after everyone has agreed to a set of changes, can you proceed to 'B' below.

   A. If everyone does NOT agree with your proposed changes, YOU need to click the 'Close pull request' button. This will close the request while you work on changes.

      Make changes on your local branch: First merge any updates that might have occurred in the meantime on the remote master branch into your local master and then merge the local master branch back into your local branch (e.g. '<your initials>fixWarnings'). Make fixes on your local branch, and commit them often and push them to the remote copy of your branch often. Once you have completed this work start again from Tutorial 4, step 1.

   B. Once you receive a thumbs-up from everyone, meaning they agree to your proposed changes, YOU need to click the 'Merge pull request' button. This will perform the merge and add your changes to the master branch. It will take you to another form where you need to add a comment about the merge, for example that everyone agreed to it. Then click the 'Confirm merge' button. This goes back to the Conversation tab for this merge. You can click on the '<> Code' tab at the top of the window to get to the code in the GitHub repository.

5. Go back to Eclipse, switch to the 'master' branch and merge the remote version into your local branch to update your local copy. In this case when the 'Merge master' window is displayed, the local master should be checked, and you need to click the 'Remote Tracking' section, 'origin/master' and perform the merge. You should choose the 'Commit' radio button for this merge.