Introduction to Groovy-Eclipse Dev

- What is Groovy-Eclipse?
- Setting Up Groovy-Eclipse Development Environment
  - Downloading Eclipse or GGTS
  - Loading Groovy-Eclipse Source Code from Git
  - Compiling Groovy-Eclipse Source Code
- Groovy-Eclipse Source Code
  - Launching Groovy-Eclipse
  - Launching Tests
- Defects
  - Defects to look at
  - Typical workflow with issues:
- Creating testcases
  - Compiler tests
- Debugging ‘groovyc’ Command
  - Attach Eclipse Debugger to ‘groovyc’ java process
  - Launch org.codehaus.groovy.tools.FileSystemCompiler as Java Application
- Where can you find the users?
- Working with Git Repository using Git Commands and eGit
  - Create a Fork
  - Creating a Branch
  - Committing Changes
  - Creating a Pull Request
  - Synchronizing Branches with Master Branch
    - Synchronizing Forked Repository with Main Repository
    - Synchronizing Branch with Master Branch
    - Pulling Changes from a Remote Repository Branch into Local Clone Repository Branch
- Coding Standards
- CI Builds
- Running Builds Locally

What is Groovy-Eclipse?

Groovy-Eclipse is a Groovy language IDE for Eclipse. Groovy is based on Java and allows mixing Groovy and Java code. Consequently, Groovy-Eclipse implementation is coupled with Java Development Tools Eclipse component (JDT)

For more information on Groovy language see
http://groovy.codehaus.org/Getting+Started+Guide

There is an article on the architecture of groovy-eclipse here, describing how JDT and groovyc are modified to join them together:
https://spring.io/blog/2009/07/30/a-groovier-eclipse-experience

Setting Up Groovy-Eclipse Development Environment

Setting up the development environment for Groovy-Eclipse requires downloading Eclipse and some extra plugins for Eclipse as well as acquiring Groovy-Eclipse source code from Git and compiling the source code.
Instructions below are based on:
http://groovy.codehaus.org/Getting+Started+With+Groovy-Eclipse+Source+Code
https://github.com/groovy/groovy-eclipse

Downloading Eclipse or GGTS

First , You'd need JDK 1.7 (Java 1.8 might work) and Eclipse 4.3.x or 3.8.x. Groovy-Eclipse supports currently Groovy 2.2.2 which is only supported for up to Java 1.7. (Java 1.8 will be supported once Groovy-Eclipse starts supporting Groovy 2.3.x)

1. Download JDK 1.7 from http://www.oracle.com/technetwork/java/javase/downloads/index.html
2. Download Eclipse 4.3.x (or 3.8.x) from http://eclipse.org/downloads/. Alternatively to skip Groovy-Eclipse download from the update site instead of Eclipse download GGTS 3.5.0 from here: http://spring.io/tools/ggts/all
3. Start Eclipse using JDK 1.7 (set -vm <path to the vm> in eclipse.ini)
4. (Skip steps 4-7 if you've downloaded GGTS 3.5.0 instead of Eclipse) In the top level menu above in Eclipse go to Help -> Install New Software…
5. In the opened Install dialog click on "Add" button (to enter a Spring Tool Suite update site to download Groovy-Eclipse)
6. Enter Name "STS 3.5.0" and Location http://dist.springsource.com/release/TOOLS/update/e4.3/ for Eclipse 4.3.x and click OK (Last path segment in the URL stands for eclipse version being used. See the list of available update sites here: http://spring.io/tools/ggts/all)
7. Find Groovy-Eclipse in the check-list, check-mark the Groovy-Eclipse entry and click "Next" button on the bottom of the dialog couple of times, accept the License Agreement and proceed with installation of Groovy-Eclipse and restart Eclipse at the end

8. (Optional) Once Eclipse/GGTS is opened in the top level menu go to Window -> Show View -> Other… and try to find a “Git” view category in the tree viewer. If there is no such category it means Eclipse Git integration is not installed and one may want to install it similarly to steps 4-7 as explained via this link: http://groovy.codehaus.org/Getting+Started+With+Groovy-Eclipse+Source+Code#GettingStartedWithGroovy-EclipseSourceCode-InstalleGit(Optional)

Loading Groovy-Eclipse Source Code from Git

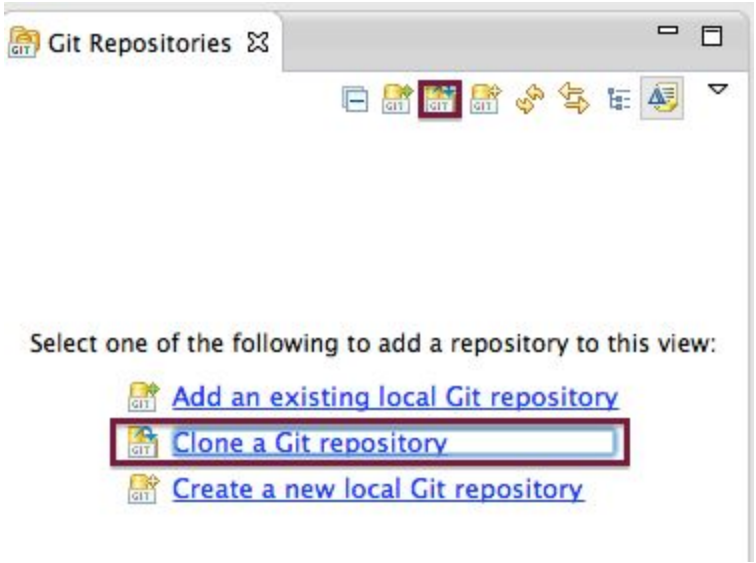
Loading Groovy-Eclipse source code can either be done from Eclipse with a help of Eclipse Git integration (eGit) or using Git command line interface and then importing the projects from a folder where the Groovy-Eclipse repository is cloned.

The Git URL for Groovy-Eclipse is https://github.com/groovy/groovy-eclipse. The page also contains info for loading the source code. Users that don’t have commit rights (new users) are advised to “Fork” the repository and create a clone of the forked repository. Forked repository URL would be https://github.com/<Git User ID>/groovy-eclipse. The forked repository can be synchronized with the original repository as described in this post: http://stackoverflow.com/questions/11646080/keep-sync-with-another-git-repository.

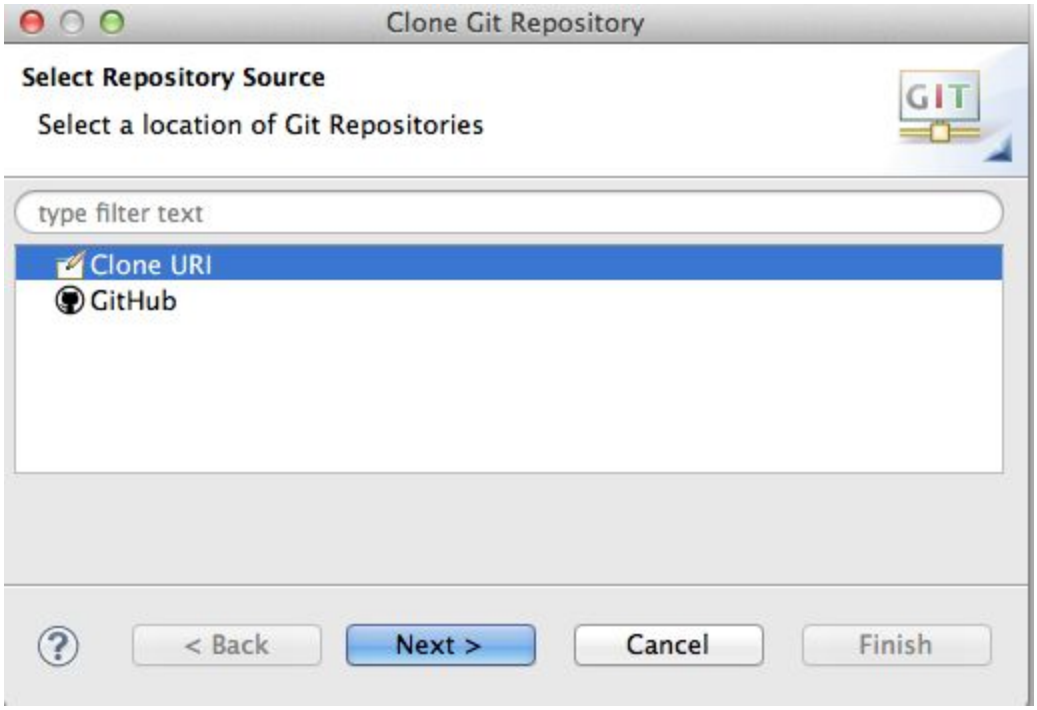
See Working with Git Repository using Git Commands and eGit section for some basic instructions on using a Git repository

Loading Groovy-Eclipse source from Git instructions using Eclipse Git integration:

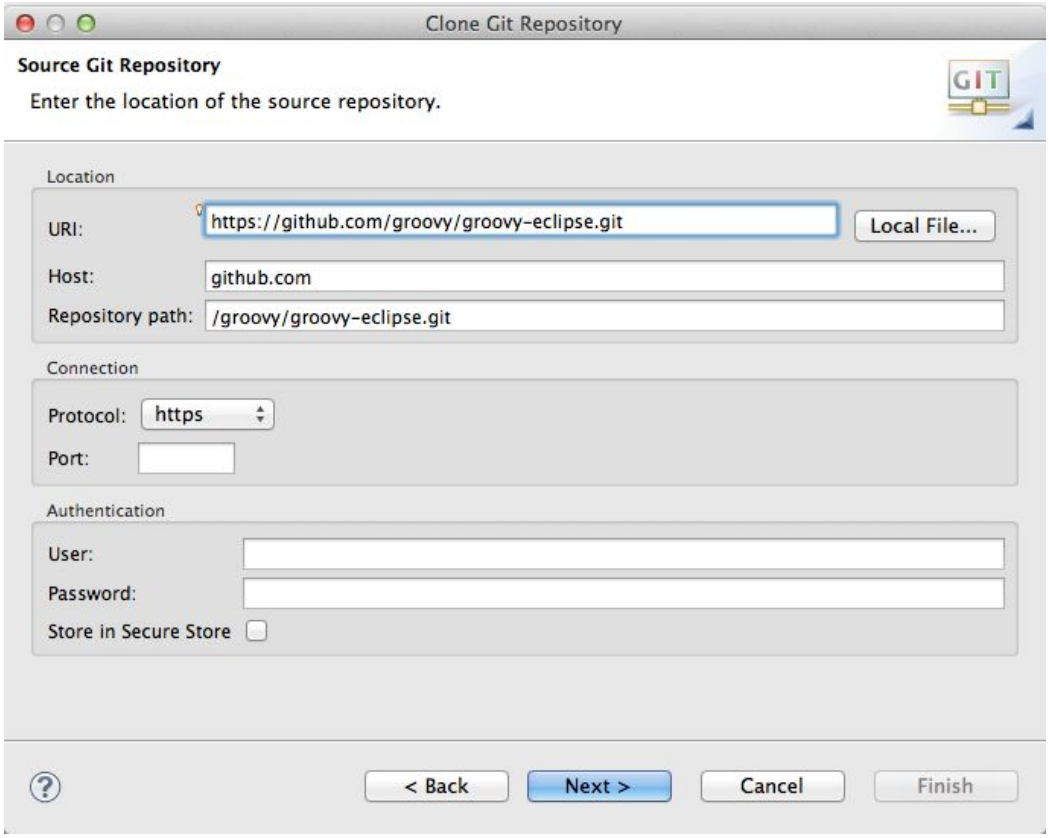
1. In Eclipse top level menu go to Window -> Show View -> Other…
2. Start typing “git” to filter out view names and select “Git Repositories” view
3. Either click on the “Clone a Git Repository” hyper link or a button in the Git Repositories” view toolbar.



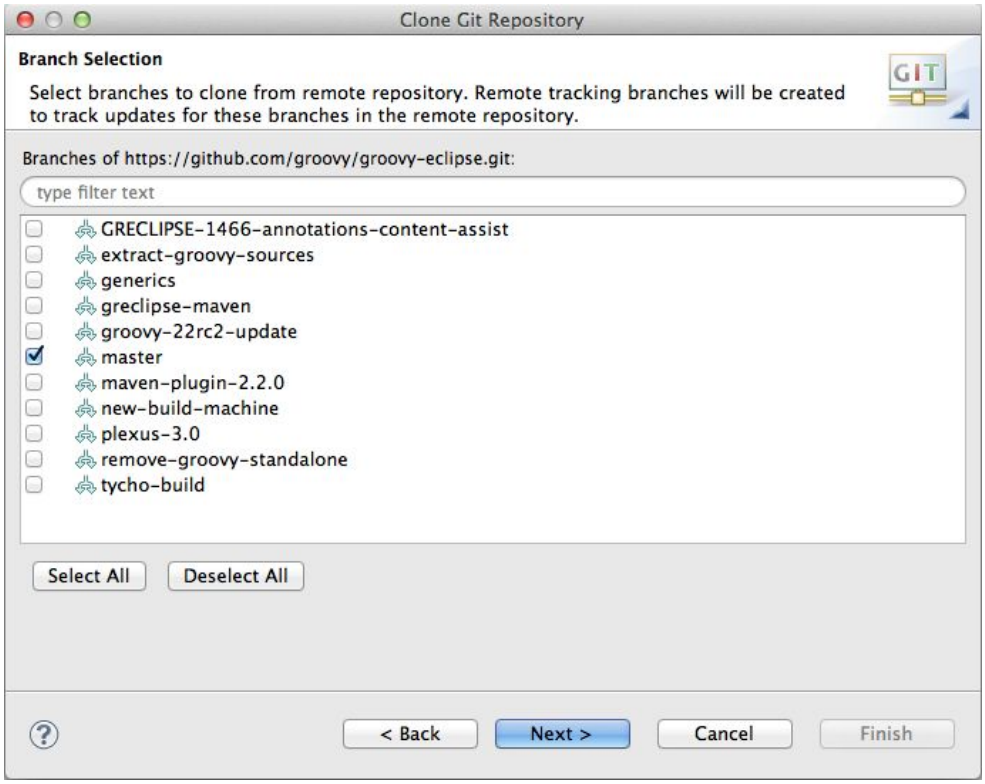
4. Select “Clone URI”.



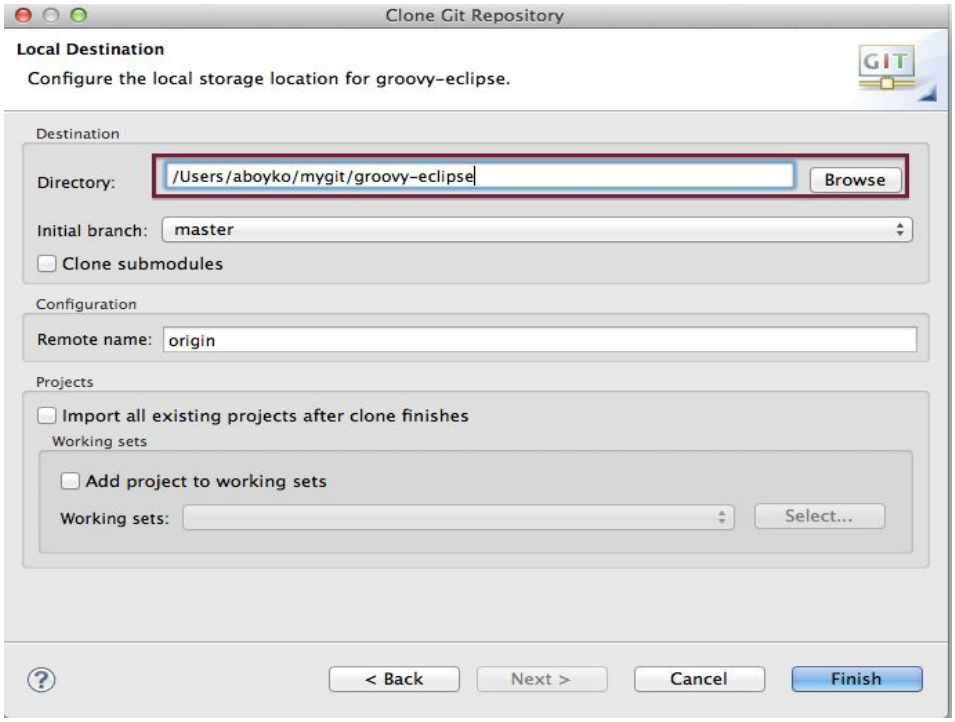
5. Copy-Paste the Git repository URL to clone in the URI field and add “.git” at the end. Most of the fields would be auto-filled. Enter the Git user id and password on the bottom.



6. Click “Next” and either select the desired branch or if not sure either select “master” or all branches.

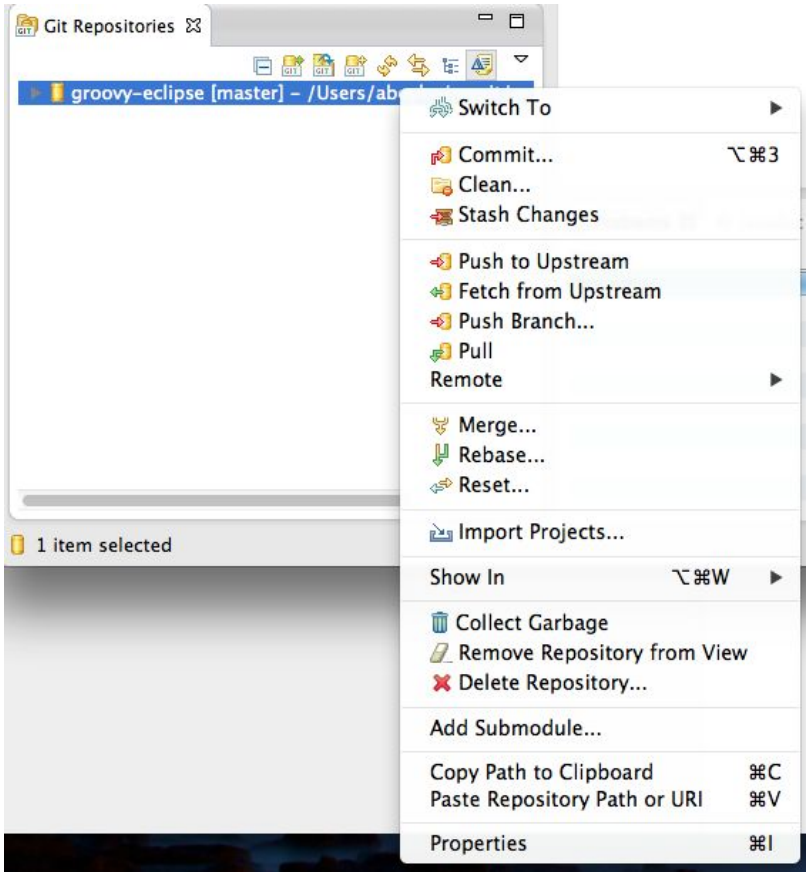


7. Enter or browse a folder on local file system to store Groovy-Eclipse Git repository clone locally.

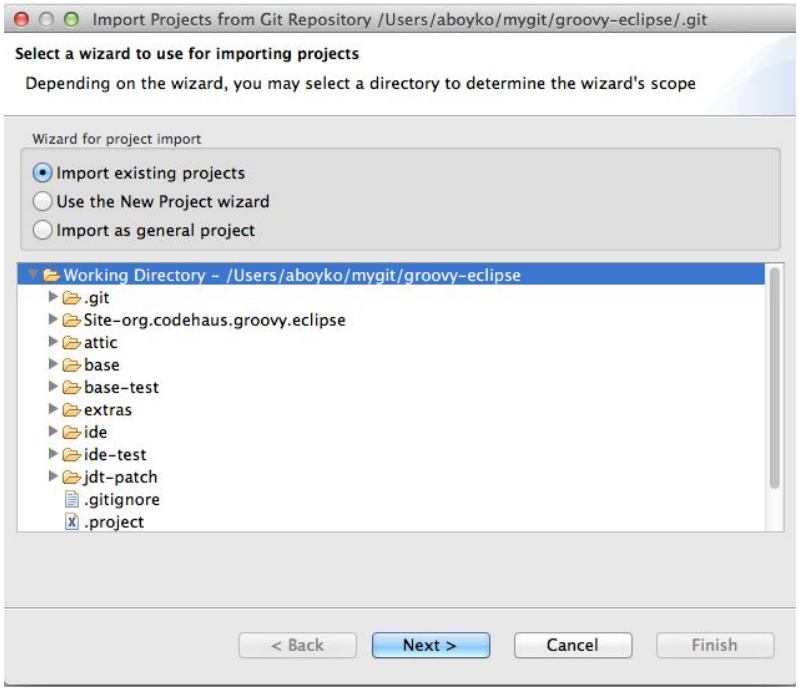


8. Click “Finish” and repository clone now appears in the “Git Repositories” view

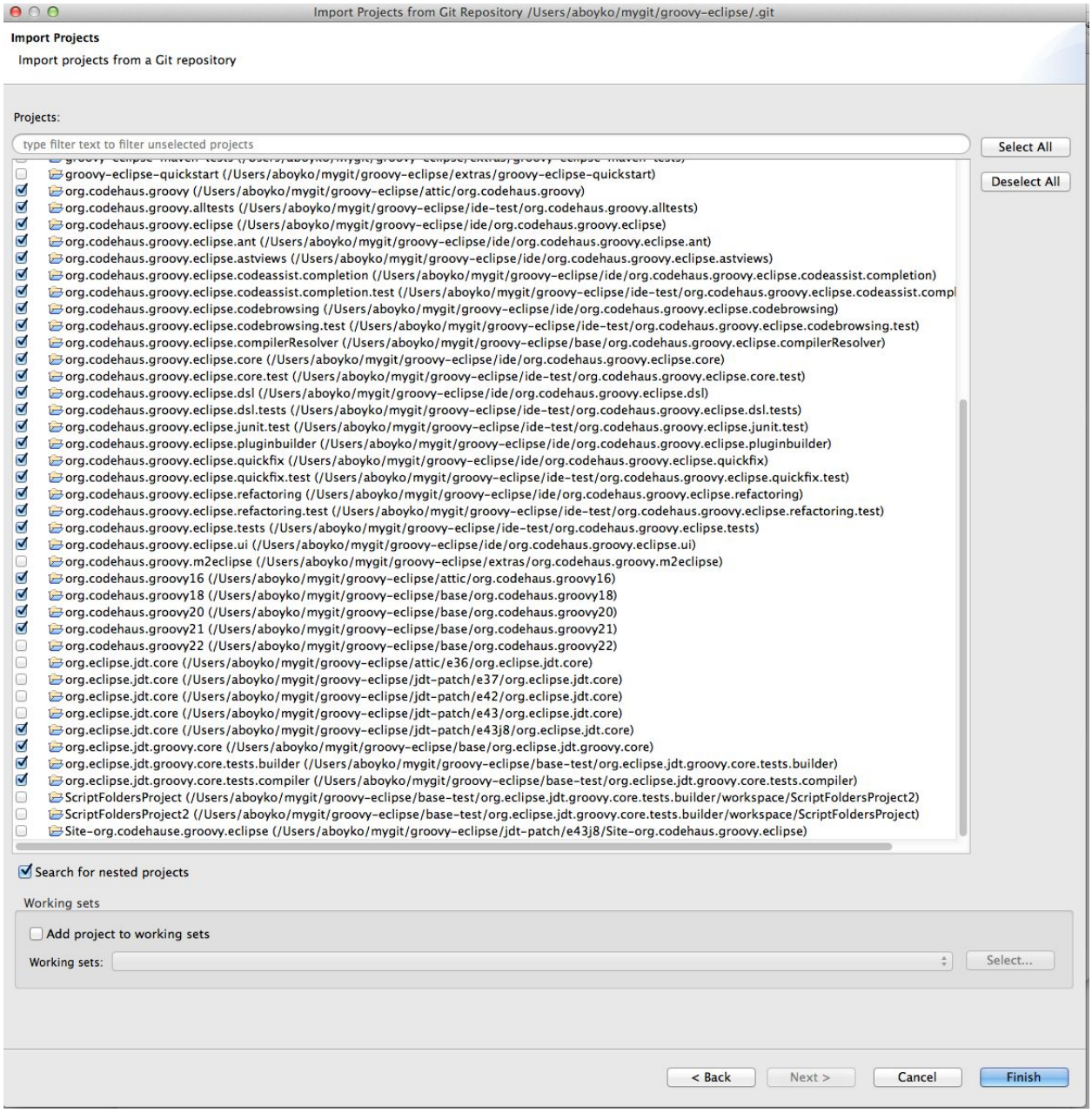
9. Right-click on the created repository and select “Import Projects...”.



10. Ensure that selections in the displayed wizard page are as shown below and click “Next”.



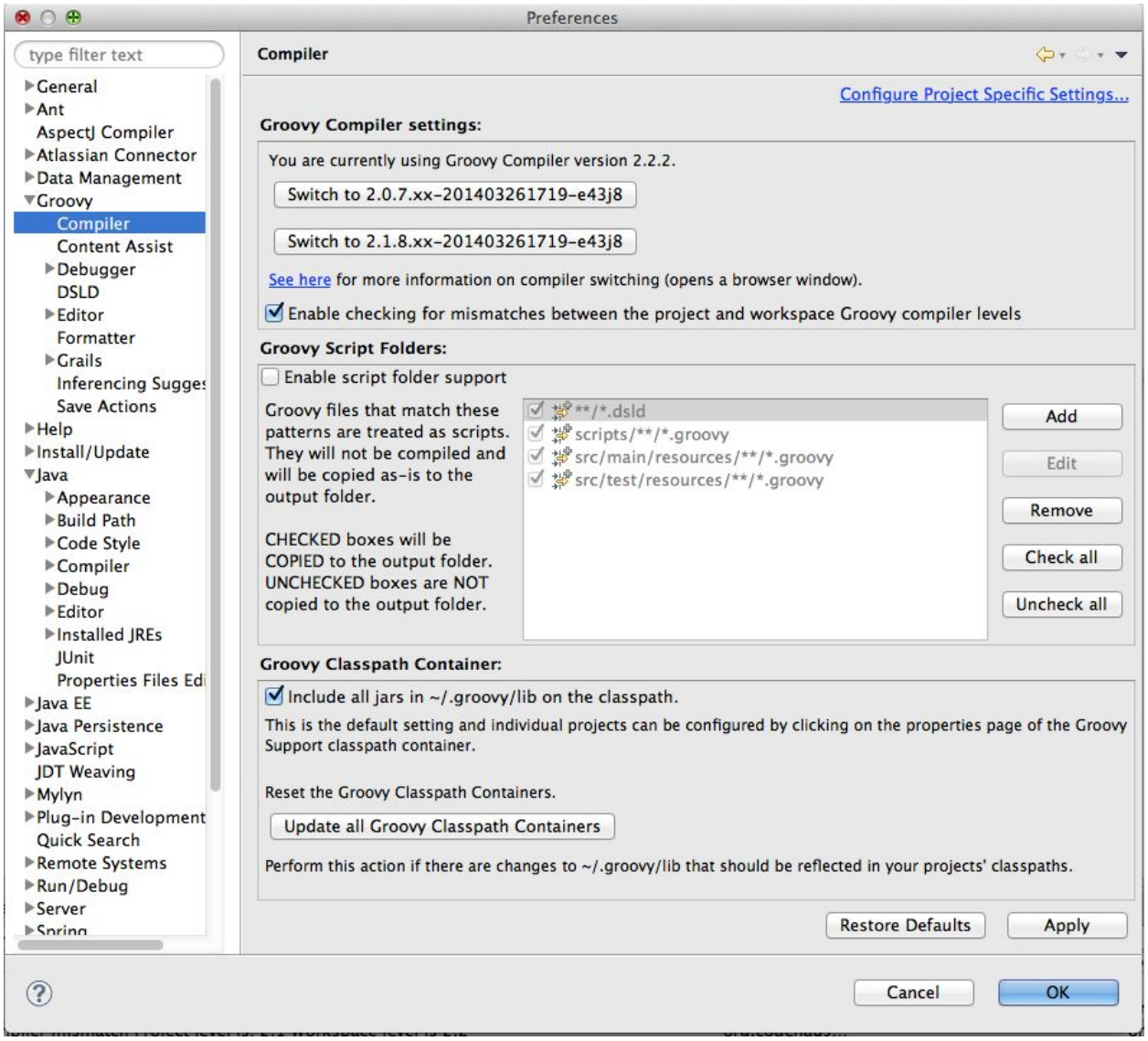
11. Next page lists projects that can be imported. Typically not all projects need to be imported as some of them are rather optional parts of Groovy support for Eclipse. More information on projects can be found either here https://github.com/groovy/groovy-eclipse (point 4) or here https://web.archive.org/web/20150102210202/http://groovy.codehaus.org/Getting+Started+With+Groovy-Eclipse+Source+Code#GettingStartedWithGroovy-EclipseSourceCode-ImportGroovy-Eclipsesourcecodeintoyourworkspace(thecomplicatedway). Otherwise, a typical set of projects is shown below.



Compiling Groovy-Eclipse Source Code

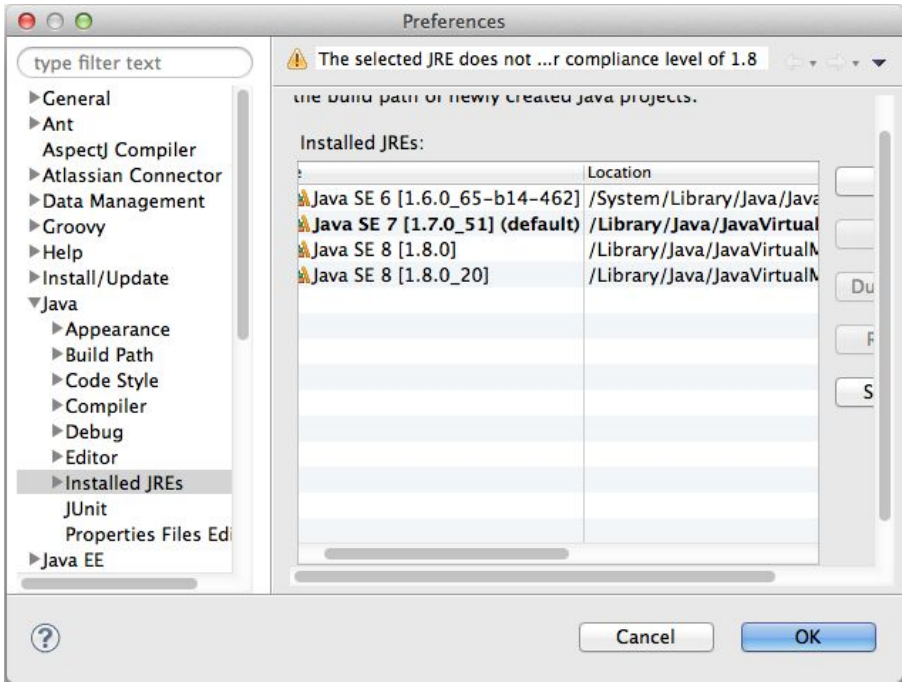
At this point the minimal set of Groovy-Eclipse projects has been loaded from the Git repository and imported into Eclipse workspace.
If all projects compiled right after importing them into Eclipse please skip this section.

There are exclamation mark decorations in Project Explorer on projects and there are errors such as “Groovy: compiler mismatch Project level is: 2.1 Workspace level is 2.2” in the “Problems” view.
Go to Eclipse Preferences -> Groovy -> Compiler and verify that current Groovy compiler is 2.1.x and not higher. Currently by default the Groovy compiler is 2.2.2 for a fresh workspace. Switch it to 2.1.8 for example by clicking the corresponding button and then re-starting Eclipse.



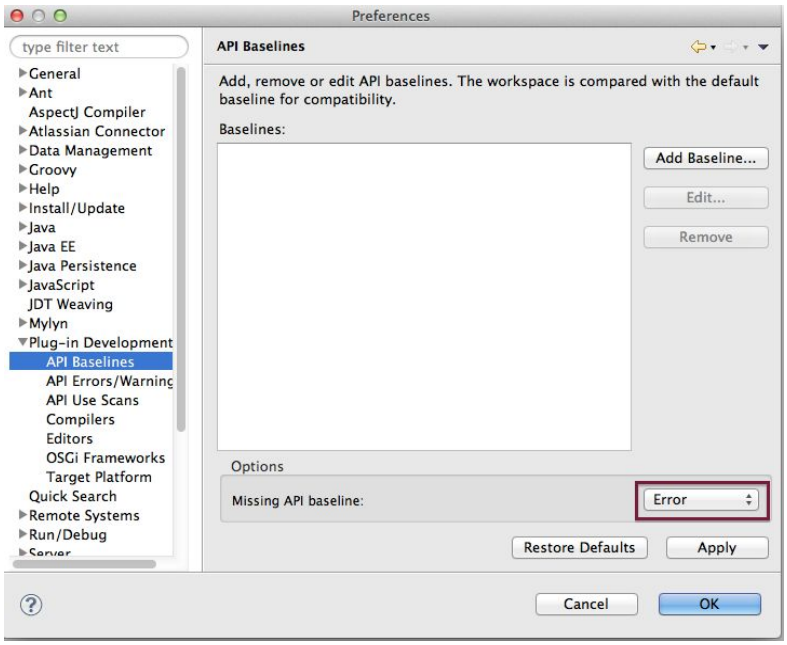
There are errors in Java5.java resource in org.codehaus.groovy16 and/or other org.codehaus.groovyXX projects.

Go to Eclipse Preferences -> Java -> Installed JREs and check-mark Java 1.7 (add it if it's not in the list by browsing to Java 1.7 on the local system)

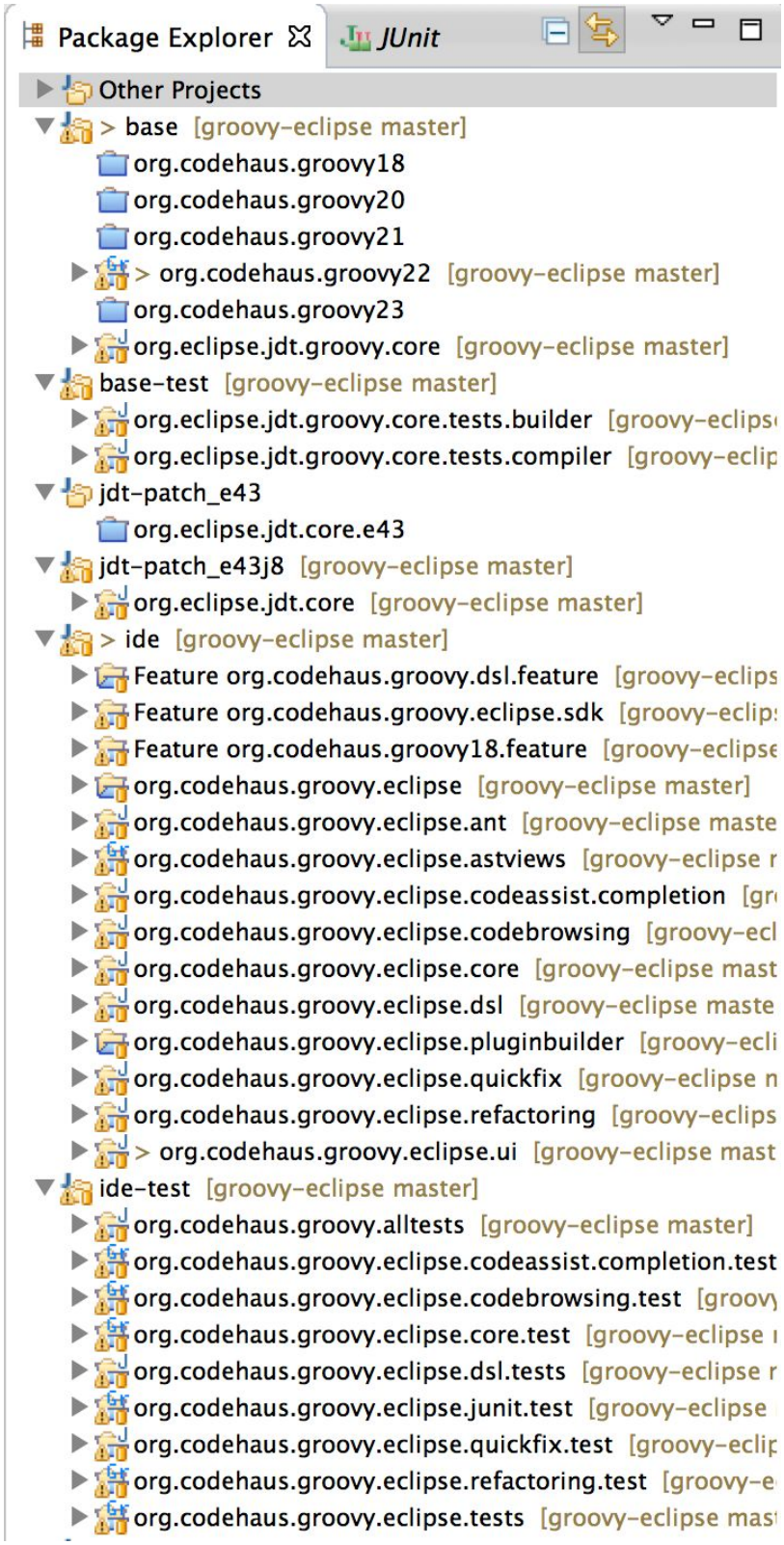


There are errors such as "An API baseline has not been set for the current workspace"

Go to Eclipse Preferences -> Plug-in Development -> API Baselines and select either Ignore or Warning instead of Error in the drop-down highlighted below. Click "OK"



Example of the projects in a working setup:



Notes about the setup:
- I have all the compiler versions imported in the ‘base’ group but I only have one of them open, the one I’m interested in at the moment.
- I have two versions of the JDT patch in here, but the open one is the org.eclipse.jdt.core.e43j8 one - this is the Eclipse Kepler Java8 patch. This means when I launch a runtime workspace I will be getting this version of JDT to underpin everything else (it will be replacing the org.eclipse.jdt.core in my current eclipse)

Groovy-Eclipse Source Code

Groovy-Eclipse is a Groovy language tooling (similar to JDT, CDT eclipse components) integrated into Eclipse. Groovy-Eclipse source code is a set of Eclipse plug-in projects, where each project is contributing Groovy tooling logic into Eclipse via various Eclipse extension points. For example Groovy specific launch configurations, compiler, debugger, editor for .groovy files etc. Each plug-in project is responsible for some specific specific features of Groovy language tooling support in Eclipse.

org.eclipse.jdt.core
This project is the so-called JDT patch. Essentially, it is our own version of the JDT plug-in from Eclipse that hooks Groovy core low level support for Groovy language into JDT since JDT does not provide extension support for this kind of integration. See LanguageSupportFactory class references (and other classes from “groovy” folder under the project) in the project to locate integration points. Issues noticed in Java tooling may sometimes be caused by Groovy-Eclipse because of this project.

org.eclipse.jdt.groovy.core
This project contains Groovy core logic for parser, compiler, type inferencing, type look ups etc. Logic here is being hooked to JDT via *org.eclipse.jdt.core* plug-in. Part of the JDT patch

org.codehaus.groovyXX
Implementations of Groovy language, where XX stands for Groovy language version. These projects are the source code for Groovy language and are there in the workspace to make Groovy-Eclipse compile. One can test Groovy-Eclipse for a specific version of Groovy by keeping the desired Groovy language version project opened and the rest closed. If anyone is tempted to change anything in any of these projects either talk to Groovy team folks or raise a defect against Groovy language here: http://jira.codehaus.org/browse/groovy

org.codehaus.groovy.eclipse.compilerResolver
Supports ability to detect and switch between available version of Groovy available within the Eclipse. Contains core logic, no UI.

org.codehaus.groovy.eclipse.junit.test
Integrates JUnits written in Groovy into Eclipse’s JUnit framework

org.codehaus.groovy.eclipse.ant
Ant integration for building classes from Groovy code.

org.codehaus.groovy.eclipse.core
Integration of Groovy core components into Eclipse: compiler, launcher, search, type look up, preferences. No UI contributions, just back-end logic

org.codehaus.groovy.eclipse.ui
Various contributions to Eclipse UI components. Preference pages, launch configuration wizard tabs, various UI actions, creation wizard for Groovy project, creation wizards for Groovy language artifacts, type browsing, search etc.

org.codehaus.groovy.eclipse.refactoring
Feature of Groovy source editor (.groovy files editor). Integration of code refactorings for Groovy language into Eclipse. (Right-click on a Groovy statement Source -> Refactor)

org.codehaus.groovy.eclipse.codeassist.completion
Feature of Groovy source editor (.groovy files editor). Integration of code completions for Groovy code. <Ctrl> + <Space> behavior for uncompleted Groovy statements

org.codehaus.groovy.eclipse.quickfix
Feature of Groovy source editor (.groovy files editor). Integration of quick fix suggestions for Groovy language. Can either be activated by selecting a statements and pressing <Ctrl> + 1 or by clicking on the error annotation in the .groovy file editor's overview ruler

org.codehaus.groovy.eclipse.dsl
Support for Groovy based Domain Specific Language. Provides contents assist, type inferencing etc. for a DSL defined by DSL descriptor

Information on other projects can be found at:
https://web.archive.org/web/20150102210202/http://groovy.codehaus.org/Getting+Started+With+Groovy-Eclipse+Source+Code#GettingStartedWithGroovy-EclipseSourceCode-ImportGroovy-Eclipsesourcecodeintoyourworkspace(thecomplicatedway)

Launching Groovy-Eclipse

Groovy-Eclipse is a set of plug-in projects for Eclipse that embed Groovy language tools into Eclipse. Consequently, Groovy-Eclipse can only be run in the context of Eclipse application. Create a new Eclipse Application launch configuration and launch it to test Groovy-Eclipse code modifications. Eclipse UI for Groovy is very similar to Eclipse Java UI. Create a new Groovy Project, create a package inside the project and then create a Groovy class inside the package (Follow New -> Other…)

Launching Tests

Almost every eclipse plug-in project from the Groovy-Eclipse code base has a JUnit test plug-in project. Check if a test project has AllXXXTests class where XXX typically stands for the component name, right-click it and click Run As -> JUnit Plug-in Test. If there is no such class right-click the project and click Run As -> JUnit Plug-in Test.
Additional information on running Unit tests can be found at:
http://groovy.codehaus.org/Getting+Started+With+Groovy-Eclipse+Source+Code#GettingStartedWithGroovy-EclipseSourceCode-Runthetests

Defects

Bug reports can be either found or filed here http://jira.codehaus.org/browse/GRECLIPSE. One would need to signup to *jira.codehaus.org* to be able to comment on JIRA's. Andy can grant admin rights that add permissions to assign, triage and resolve JIRAs.

There are a number of areas in Groovy-Eclipse requiring work such as:

- Groovy Parser reliability for not well-formed Groovy source (file is being edited)
- Static Compilation (@CompileStatic)
- Groovy Debugger
- Groovy source formatter
- Generic Types and Enums (JIRAs for Enums might be fixed indirectly)

Defects to look at

Some easy defects to start with:
~~http://jira.codehaus.org/browse/GRECLIPSE-1522~~ ~~(If fixed indirectly needs a Junit)~~
~~https://jira.codehaus.org/browse/GRECLIPSE-1538~~ ~~(If fixed indirectly needs a Junit)~~
~~http://jira.codehaus.org/browse/GRECLIPSE-1443~~
~~http://jira.codehaus.org/browse/GRECLIPSE-1528~~
~~http://jira.codehaus.org/browse/GRECLIPSE-1693~~
http://jira.codehaus.org/browse/GRECLIPSE-1696
http://jira.codehaus.org/browse/GRECLIPSE-1703
http://jira.codehaus.org/browse/GRECLIPSE-1702
http://jira.codehaus.org/browse/GRECLIPSE-1691

Some slightly more difficult:
https://jira.codehaus.org/browse/GRECLIPSE-1531
http://jira.codehaus.org/browse/GRECLIPSE-1426
http://jira.codehaus.org/browse/GRECLIPSE-1525
http://jira.codehaus.org/browse/GRECLIPSE-1351
http://jira.codehaus.org/browse/GRECLIPSE-1348

http://jira.codehaus.org/browse/GRECLIPSE-1527
http://jira.codehaus.org/browse/GRECLIPSE-1682

Some features:
http://jira.codehaus.org/browse/GRECLIPSE-884
http://jira.codehaus.org/browse/GRECLIPSE-1273
http://jira.codehaus.org/browse/GRECLIPSE-1619

Note: All the above (and more) issues are now tagged with 'help-requested' in the Jira tracker. A live list of open 'help-requested' issues is here.

Typical workflow with issues:

- Ensure you have a Jira account at codehaus (https://jira.codehaus.org/secure/Dashboard.jspa)
- You can then comment on issues where you want to collect more information from the raiser. If they don't provide enough information it is reasonable to ask for sample code that demonstrates the problem.
- Once you find an issue you want to work on, assign it to yourself (in a few weeks we might start assigning them to you).
- We try to practice test driven development so create a test that should initially fail, then fix it up.
- Create a pull request containing the test and the fix and submit it. One of the existing committers may have some comments on it where you need rework some things, so there may be a need for further edits and to submit a new pull request.
- Add a note to the groovy eclipse jira pointing to the pull request.
- Once the pull request is accepted, the person who accepts it might update the jira to indicate the problem is fixed but if they forget, you can also do that. Close the jira as fixed.

Creating testcases

- There are many kinds of testcases depending on the kind of issue.

Compiler tests

If you have a snippet of code that just doesn't compile cleanly, then you should capture it as a compiler test. There are two test files that contain compiler tests:

`GroovySimpleTest`

For the most simple cases. Look at the test methods in here, there is either a call to 'runConformTest()' if the snippet should successfully compile and run, or 'runNegativeTest()' if it should not compile (in which case it polices the error messages that will come out). Each test is simply the testcode captured as a string and then expected output or expected errors. It is possible to pass multiple source files to these run methods.

`BasicGroovyBuildTests`

These are more about creating tests that represent a typical build flow running against a project. These tests let you create a project, then add files to it (groovy or java) and then call fullBuild/incrementalBuild to simulate what would happen in a real eclipse and then either police the expected errors or expected output. If you have an issue that says it is failing on an incremental build (but not a full build) you might be creating a BasicGroovyBuildTest.

TODO: What you could do to get familiar with these tests is write some for new functionality. For example I just committed the org.codehaus.groovy23 bundle. This should have added traits support but I haven't tested it! The documentation is here: http://beta.groovy-lang.org/docs/groovy-2.3.0-beta-1/html/documentation/core-traits.html it would be good to add some of those samples as regression tests - probably into GroovySimpleTest above. These tests will need to check the compiler level of course and exit early if not at groovy 2.3 level. In order to track this work perhaps create a jira against the groovy-eclipse project called 'Add testcases for traits'.

Debugging 'groovyc' Command

Often defects are addressing something that doesn't compile in Groovy Eclipse and yet compiles fine when compiled with a 'groovyc' command executed from command line interface. Such defects would require one to investigate differences between 'groovyc' command compilation (pure Groovy) and Groovy Eclipse compilation. This involves debugging 'groovyc' command execution. There are 2 ways debugging 'groovyc' command:

- Attach eclipse debugger to 'groovyc' java process
- Launch org.codehaus.groovy.tools.FileSystemCompiler as a Java Application

It is recommended to have Groovy source from https://github.com/groovy/groovy-core in your workspace as code in Groovy Eclipse org.codehaus.groovy doesn't exactly match the original groovy code and has a few Groovy Eclipse specific fixes.

Attach Eclipse Debugger to 'groovyc' java process

1. Execute the following in the console export JAVA_OPTS="-Xdebug -Xnoagent -Djava.compiler=NONE -Xrunjdwp:transport=dt_socket,address=5000,server=y,suspend=y"
2. Start 'groovyc' command from the console, e.g. '<path to groovyc>/groovyc <groovy files to compile>. Process should bu suspended until debugger is attached.
3. Create a new 'Remote Java Application" launch configuration in Eclipse. Specify project from the workspace corresponding the version of Groovy for the ran 'groovyc' command, leave host as localhost and specify the port 5000
4. Launch the created "Remote Java Application"

Launch org.codehaus.groovy.tools.FileSystemCompiler as Java Application

1. Create a new "Java Application" launch configuration in Eclipse
2. Specify the project from the workspace corresponding the required Groovy version. Specify org.codehaus.groovy.tools.FileSystemCompiler as the Main class to launch
3. Specify groovy files to compile (absolute path) on the "Arguments" tab of the launch configuration dialog in the "Program Arguments" text box
4. Run the newly created launch configuration

Working with Git Repository using Git Commands and eGit

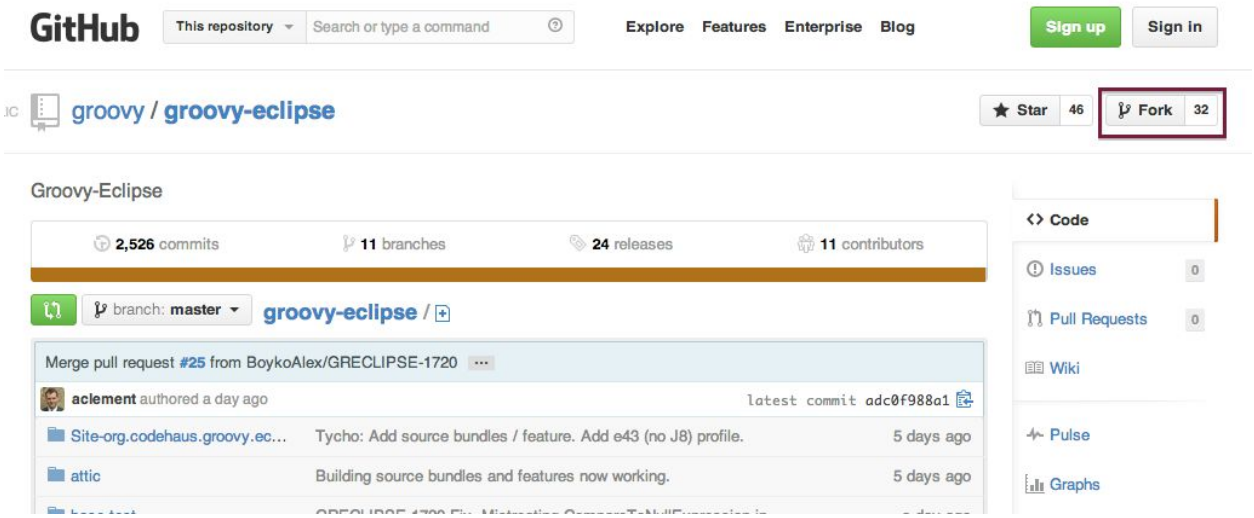
Git is a cloud hosted source control facility. This section contains instructions on some basic interactions with a Git repository such as forking a repository, creating branches, committing changes and synchronizing the code between branches. Prerequisite for doing any work with a Git repository on GitHub is creating a GitHub account.

Usually if a user has commit rights work can be performed on the main branch. However, it is recommended to create a separate branch for every feature or defect work.
If user doesn't have commit rights then the main branch commits are not allowed and a "Fork" of a repository needs to be created. Forked repository is sort of a user's private repository based on the original repository.
Synchronization of branches with the main branch is required to get the latest changes from the main branch into the current.
Committing changes can be done directly to the main branch or via branches by creating pull requests which could later be merged to the repository's main branch.

Git repository used below is a Groovy-Eclipse repository:
https://github.com/groovy/groovy-eclipse

Create a Fork

Paste the Git repository URL in the browser and open the page. Locate the “Fork” button on the page and click it.

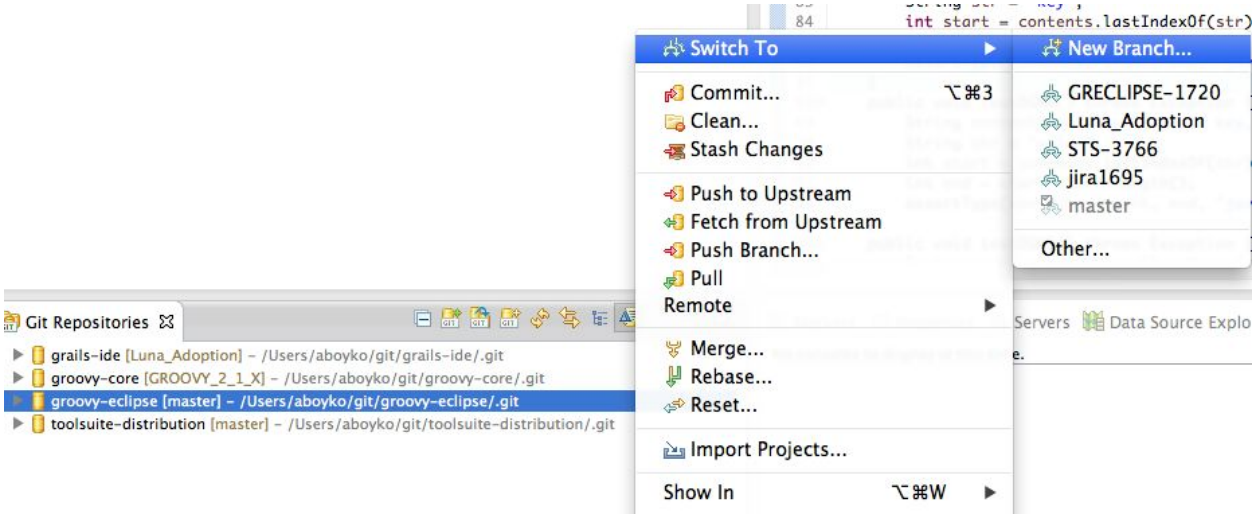


After performing the Fork operation the repository URL in the browser would become https://github.com/BoykoAlex/groovy-eclipse (BoykoAlex being my GitHub user id). This URL is to be used for loading the source code instructions for this are in Loading Groovy-Eclipse Source Code from Git steps 1-8.

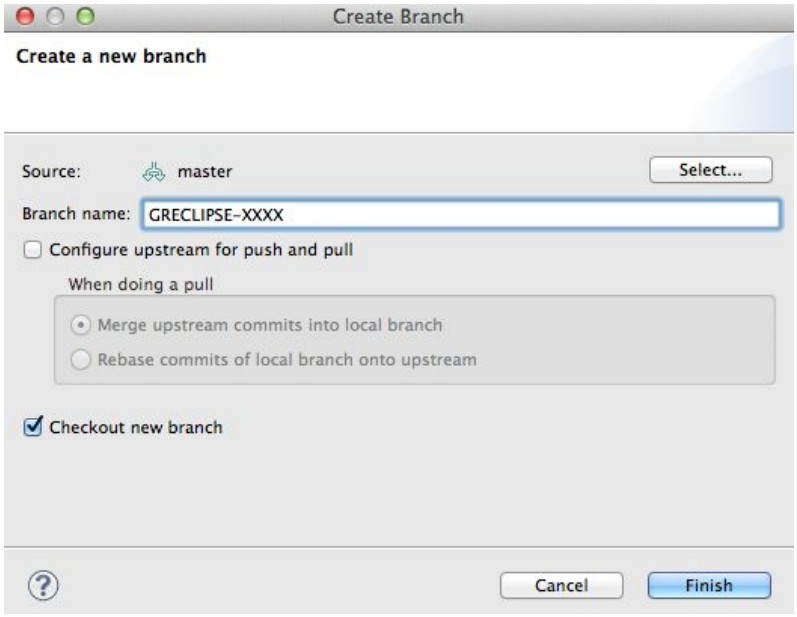
Creating a Branch

Branch is recommended to be created for work on a single work item or a series of related work items that are meant to be delivered together.

The easiest way to create a branch is via Eclipse’s Git integration. Right-click on the cloned repository in the Eclipse Git Repositories view and navigate to “Switch To…” menu group. One could either click “New Branch” menu item right away that takes to branch creation wizard or click “Other…” which takes to branches explorer that provides create/delete/edit branches capabilities.



Enter a self-explanatory name for the branch. For example a JIRA id: GRECLIPSE-XXXX and click Finish.



Note that “Checkout new branch” is checked, which implies that code in the workspace would be coming from this new branch, i.e. annotations next to project would be switched to GRECLIPSE-XXXX.

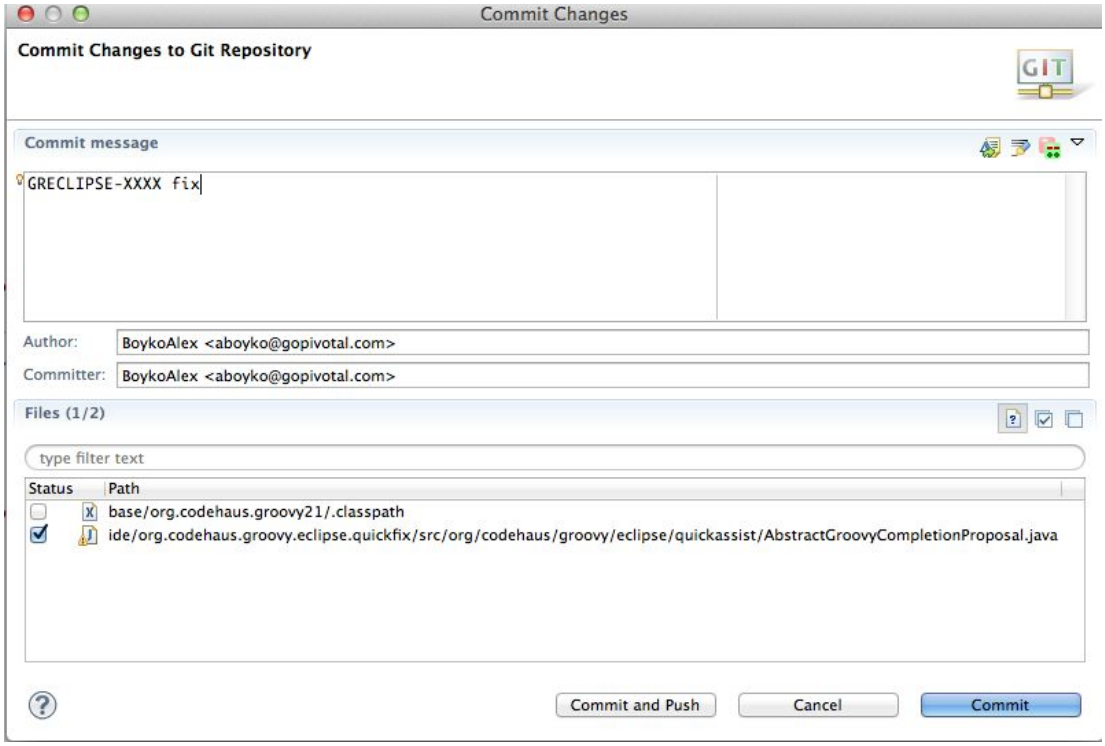
At this point the branch is created locally. In order to push it to a remote Git repository, right-click on the repository in the Git Repositories view and click on “Push Branch…”. Keep default settings in the wizard and click Next and eventually Finish.

Deleting a branch can be done similarly to creating via the cloned repository context menu navigate to Switch To… -> Other… Select the branch and click “Delete” button. Before deleting a branch make sure it’s not currently checked out in the workspace. For example to delete the

created GRECLIPSE-XXXX branch one would need to switch to “master” branch for instance and then try deleting the GRECLIPSE-XXXX branch.

Committing Changes

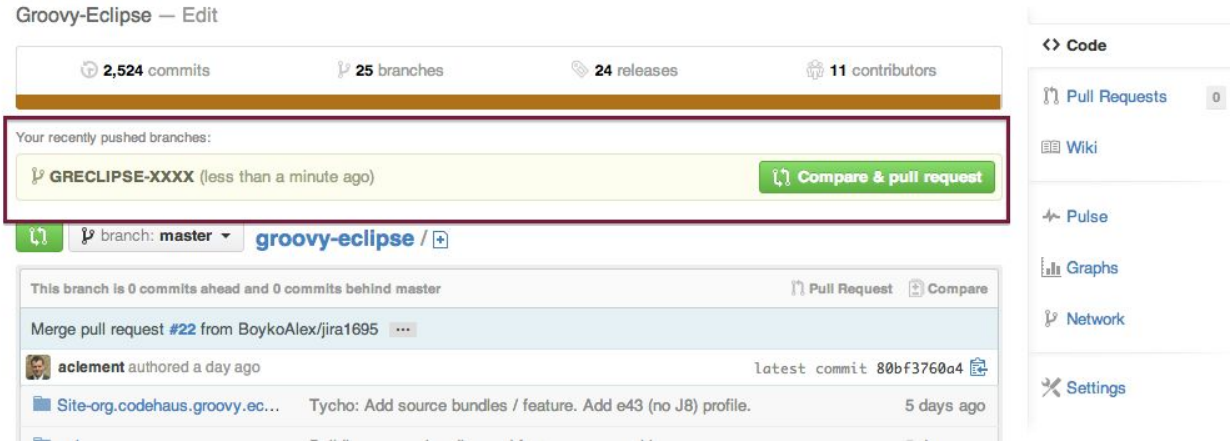
Once contents of the projects managed by a Git repository are modified affected files will be marked with “>” in the Project Explorer and/or Package Explorer.
Commit changes by right-clicking on any projects from the repository in the Project Explorer view. Navigate to Team -> Commit… Select resource changes to which should be committed, enter commit message etc.



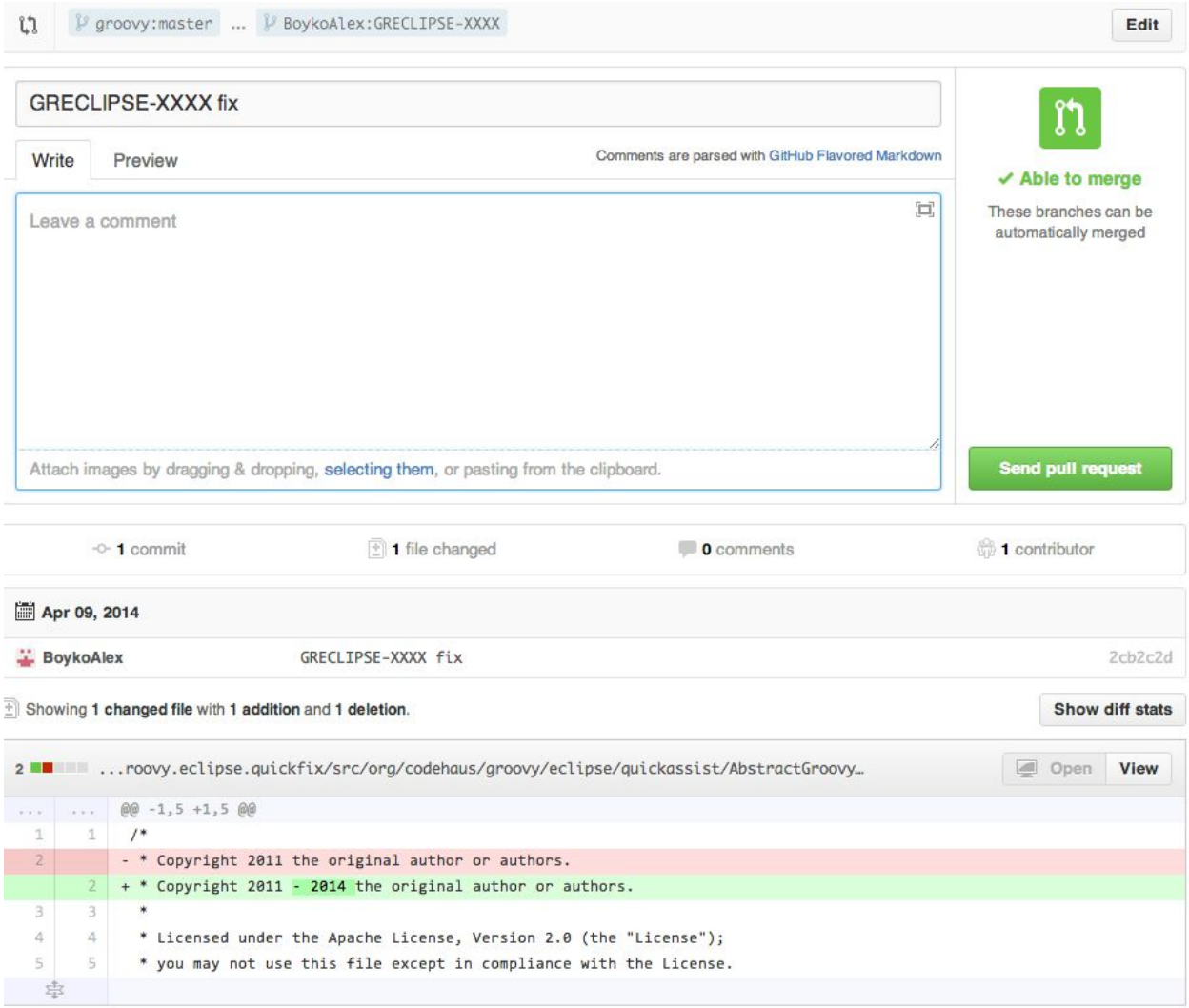
“Commit and Push” would commit changes in your local cloned repository branch and push these changes to the remote branch. Commit would just commit without pushing to the remote repository. (In this case pushing can be done separately later on). Typically one would perform “Commit and Push”

Creating a Pull Request

Pull Requests need be created to merge commits in branches with the master repository branch. (Consequently, no need for a pull request if code is committed and pushed to the master repository branch)
Code is committed and pushed to a branch with repository URL https://github.com/BoykoAlex/groovy-eclipse. Navigate to this URL in the browser. Note that the content of the page has changed with the information about the new commit. It’s marked with a red rectangle. Click on the “Compare & pull request” button.



The displayed page would allow to finalize a pull request. A comment can be entered and code changes reviewed. Click “Send Pull Request” to submit the pull request



Changes will be reviewed by committers, merged with the master branch and the merge notification email sent out.

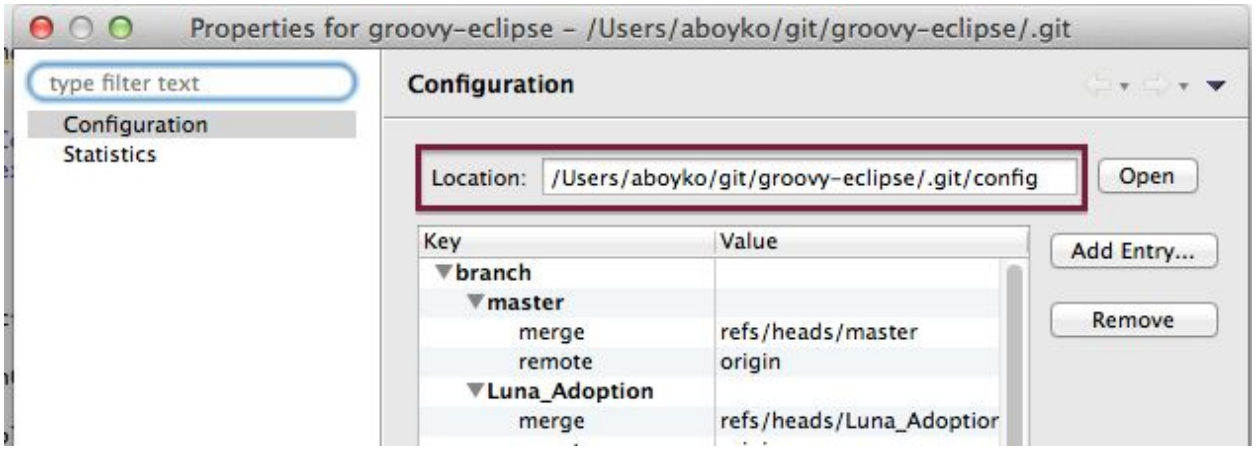
Synchronizing Branches with Master Branch

While working on a branch other commits might be occurring into the master branch. These changes need to be propagated into the branch being worked on.

If the repository's master branch is forked then the forked repository (master branch of the forked repository) should be updated with the latest changes first.

Synchronizing Forked Repository with Main Repository

Right-click the repository in the Git Repositories view and click on Properties. Copy the location of the local clone of the repository.



The path to copy would be */Users/aboyko/git/groovy-eclipse*
Open the Terminal or Command Prompt navigate to that folder and execute the following command to the main repository as a remote:
git remote add upstream ***https://github.com/groovy/groovy-eclipse.git***
Now pull the changes from the main repository onto the forked repository with the following command:
git pull upstream master
(Both of these commands can be performed with Git integration for Eclipse too)

Synchronizing Branch with Master Branch

It is assumed that the branch is currently checked out. Right-click the cloned repository in the Git Repositories view and click on "Merge...". Then in the tree viewer select the branch to merge the changes from.

Pulling Changes from a Remote Repository Branch into Local Clone Repository Branch

If changes are expected to occur to a branch being worked on then pulling these changes into local clone branch can be performed by right-clicking the repository in the Git Repositories view and clicking "Pull" menu item. (Example: working with the master branch of the repository)

Coding Standards

We are not strictly enforcing anything. Try to ensure your code fits the style of what is there.

CI Builds

The automated builds use maven tycho to compile all the code run tests and upload build artefacts to update sites. The CI build results are available here:
https://build.spring.io/browse/GRECLIPSE

For each target platform (E37, E42, E43, E44) there are two builds:

- EXX Build, test and publish
- EXX Build and publish

The only difference between those builds is that one of them runs regression tests while the other does not. The builds which 'skip tests' only run when manually triggered, the 'test' builds trigger automatically when changes are committed to master.

These are the different 'target platforms': E37, E42, E43, E43-j8, E44.
E43-j8 is E43 with Java 8 support patch installed.

Running Builds Locally

It is possible to run the builds locally on your own machine but you will not be able to 'deploy'. Deploying requires secret acceskeys that exist only on the CI build hosts. Note also that normally these builds run on Unix-based (Mac/Linux) systems. They have never been run on Windows machines. So we do not know if running them on Windows will work.

To run the build you need a recent version of maven. The build server uses Maven 3.1.1.

From the root of the git working directory run a command like the following:

```
mvn -Pe44 clean install
```

This will build greclipse for Eclipse 4.4 and run all the regression tests. Replace -Pe44 with another profile (-Pe37, -Pe42, -Pe43j8 etc) to build for a different target platform.

To skip the test execution:

mvn -Pe44 clean install **-Dmaven.test.skip=true**

The update site will be here: ${git-root}/Site-org.codehaus.groovy.eclipse/target/site
You can install directly from this site using it as a ‘local update site’.