### COLLABORATORS

<table>
<thead>
<tr>
<th>ACTION</th>
<th>NAME</th>
<th>DATE</th>
<th>SIGNATURE</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRITTEN BY</td>
<td></td>
<td>September 3, 2012</td>
<td></td>
</tr>
</tbody>
</table>

### REVISION HISTORY

<table>
<thead>
<tr>
<th>NUMBER</th>
<th>DATE</th>
<th>DESCRIPTION</th>
<th>NAME</th>
</tr>
</thead>
</table>
Contents

1 Introduction ................................................................. 1
  1.1 Overview ................................................................ 1
  1.2 About this manual .................................................. 1

2 Rules ............................................................................. 2
  2.1 Code Rules ............................................................. 2
  2.2 Review Rules .......................................................... 2

3 Installing and Configuring Plugins ................................. 3
  3.1 Installing the SVN Plugin .......................................... 3
    3.1.1 Installation of Subclipse (textual) ....................... 3
    3.1.2 Installation of Subclipse (graphical) .................... 4
  3.2 Installing the Git Plugin ........................................... 7
    3.2.1 Installation of EGit (textual) ....................... 7
    3.2.2 Installation of EGit (graphical) ................... 8
    3.2.3 Known issues ............................................ 10
      3.2.3.1 Password for the private key is not accepted ....... 10
  3.3 Installing the JTourbus Plugin .................................. 10
    3.3.1 Installation of JTourBus (textual) .................... 10
    3.3.2 Installation of JTourBus (graphical) ................. 11

4 Sourcecode and Review .................................................. 14
  4.1 Version Control with Git ........................................ 14
    4.1.1 Introduction to Git ......................................... 14
    4.1.2 Git clone with EGit (textual) ..................... 14
    4.1.3 Git clone with EGit (graphical) .................... 15
  4.2 Configuring Git for Gerrit ....................................... 20
    4.2.1 Creating an SSH key ..................................... 20
    4.2.2 Setting your name and email address .............. 23
  4.3 Register on Gerrit ................................................ 25
    4.3.1 Signing up using OpenID ............................. 25
List of Figures

4.1 Code, Review and Test workflow ................................................. 30
List of Tables

5.1 Destination Reference ................................................................. 48
Chapter 1

Introduction

1.1 Overview

Saros is using the Git version control system to manage the sourcecode, Gerrit for code review and Jenkins for Continuous Integration. We assume you want to help to improve Saros and contribute your modifications. This little guide should help you to get started, setup your environment, install Eclipse plugins and make your first modification and put it up for review. This guide provides the information in both text and pictures, pick the one you prefer.

Usage of Images
You might have eyed at the number of pages of this guide, don't be afraid. We are using a lot of pictures through out this guide and they take a lot of space.

1.2 About this manual

This manual is created with DocBook. The sources are managed with Git and are available from the same server as the main Saros plugin (git clone from http://saros-build.imp.fu-berlin.de/gerrit/saros-developer-guide), the review is done using Gerrit and Jenkins is used to automatically update the website.
Chapter 2

Rules

2.1 Code Rules

We have some simple rules for coding convention and commit messages. The current version can be found here. Please make sure the Saros Formatter is enabled for the Saros projects.

2.2 Review Rules

Every change needs to be reviewed. This is for catching errors before they enter the tree and for allowing your peers to learn and discover the code through review. As a courtesy to others, test your changes before asking them to be reviewed. We have unit tests and subset of end-to-end tests that can be executed quickly.
Chapter 3

Installing and Configuring Plugins

3.1 Installing the SVN Plugin

Saros can integrate with the SVN Team support and needs to have access to the plugin when compiling and being executed. We are using version 1.6.x of the Subclipse plugin.

3.1.1 Installation of Subclipse (textual)

1. **OPEN THE NEW SOFTWARE DIALOG**
   Help → Install New Software

2. **PRESS THE ADD BUTTON TO ADD A SITE**
   Add to add the site.

3. **ADD SUBCLIPSE REPOSITORY**
   Pick a name for the Name field, enter the following http://subclipse.tigris.org/update_1.6.x into the Location field and press Ok.

4. **SELECT SUBCLIPSE**
   Select the Subclipse group and press Next.

5. **REVIEW THE INSTALLATION DETAILS**
   Review the installation details and continue by pressing Next.

6. **ACCEPT LICENSES**
   Review and accept the license and continue by pressing Finish. This will start the installation

7. **INSTALLATION PROCESS**
   You will be presented the progress of the installation

8. **UNSIGNED PACKAGES**
   The packages are not cryptographically signed, nothing we can change right now.

9. **RESTART ECLIPSE**
   You will be asked to restart Eclipse at the end of the installation.

10. **FEEDBACK**
    You are asked if you want to participate in providing feedback, the choice is yours.
3.1.2 Installation of Subclipse (graphical)

1. **Open the New Software Dialog**

![Image of Eclipse IDE with the New Software Dialog open]

2. **Press the Add Button to add a site**

![Image of Eclipse IDE with the Add Software Site window open]

3. **Add Subclipse repository**

![Image of Eclipse IDE with the Subclipse repository selected]

4. **Select Subclipse**

![Image of Eclipse IDE with Subclipse selected]
5. **Review the Installation Details**

6. **Accept Licenses**

7. **Installation Process**
8. **UNIGNED PACKAGES**

9. **RESTART ECLIPSE**

10. **FEEDBACK**
3.2 Installing the Git Plugin

The Saros Team is using Git to manage the source code. You will need to install the EGit plugin to get a copy of the Saros source code and start working on it.

3.2.1 Installation of EGit (textual)

1. **Open the New Software Dialog**
   
   Help → Install New Software

2. **Press the Add Button to Add a Site**
   
   Add to add the site.

3. **Add EGit Repository**
   
   Pick a name for the Name field, enter the following [http://download.eclipse.org/egit/updates](http://download.eclipse.org/egit/updates) into the Location field and press Ok.

4. **Select EGit**
   
   Select the Eclipse Git Team Provider and press Next button.

5. **Review the Installation Details**
   
   Review the installation details and continue by pressing the Next.

6. **Accept Licenses**
   
   Review and accept the license and continue by pressing Finish. This will start the installation.

7. **Installation Process**
   
   You will be presented the progress of the installation.

8. **Restart Eclipse**
   
   You will be asked to restart Eclipse at the end of the installation.
3.2.2 Installation of EGit (graphical)

1. **Open the New Software Dialog**

2. **Press the Add Button to add a site**

3. **Add EGit repository**

4. **Select EGit**
5. **Review the Installation Details**

6. **Accept Licenses**

7. **Restart Eclipse**
3.2.3 Known issues

3.2.3.1 Password for the private key is not accepted

This issue is described in Bug 326526 and is caused by EGit/JSCH not being able to read AES encrypted private keys. The available workarounds are to use 3DES encrypted keys, set the `GIT_SSH=/usr/bin/ssh` before launching Eclipse or upgrade JSCH to version 0.1.45.

3.3 Installing the JTourbus Plugin

JTourBus allows to describe aspects of the code as route. Each route has a numbered list of stops inside the code. The route can be browsed from within eclipse using this plugin. JTourBus is used in Saros to document the basics and the following will inform you on how to install the plugin.

3.3.1 Installation of JTourBus (textual)

1. **OPEN THE NEW SOFTWARE DIALOG**
   
   Help → Install New Software

2. **PRESS THE ADD BUTTON TO ADD A SITE**
   
   Add to add the site.

3. **ADD JTOURBUS REPOSITORY**
   
   Pick a name for the Name field, enter the following `http://saros-build.imp.fu-berlin.de/update-jtourbus` into the Location field and press Ok.

4. **SELECT JTOURBUS**
   
   Select JTourBus and press Next button.

5. **REVIEW THE INSTALLATION DETAILS**
   
   Review the installation details and continue by pressing the Next.

6. **ACCEPT LICENSES**
   
   Review and accept the license and continue by pressing Finish. This will start the installation.

7. **ACCEPT UNSIGNED CONTENT**
   
   Press Ok to install unsigned content.
8. **INSTALLATION PROCESS**
   You will be presented the progress of the installation

9. **RESTART ECLIPSE**
   You will be asked to restart Eclipse at the end of the installation.

### 3.3.2 Installation of JTourBus (graphical)

1. **OPEN THE NEW SOFTWARE DIALOG**

   ![Screenshot of Eclipse new software dialog](image)

2. **PRESS THE ADD BUTTON TO ADD A SITE**

   ![Screenshot of adding a site](image)

3. **ADD JTOURBUS REPOSITORY**

   ![Screenshot of adding repository](image)
4. **Select JTourBus**

5. **Review the Installation Details**

6. **Accept Licenses**
7. **ACCEPT UNSIGNED**

8. **RESTART ECLIPSE**
Chapter 4

Sourcecode and Review

We are using Git to manage our source code, Jenkins for continuous integration and Gerrit for the code review. We assume that you understand the benefits of version control, continuous integration and code review but you might not know the specific utilities. This chapter should help you find out how they are integrated and how you are going to use them.

You will need to get the source code from our main git repository. After you have completed modifications to the source you can create local commits. Your commits can be pushed to a repository you own or you can push them into the Gerrit code review system. This will result in Gerrit creating a Chase and will lead to the Jenkins server being triggered and asked to compile and test your commits. The result will be returned to Gerrit. Your teammates are asked to make reviews of your change. You are asked to modify your change until both the Jenkins result and the feedback of your teammates is positive. Your commits will be automatically integrated into the main repository.

4.1 Version Control with Git

4.1.1 Introduction to Git

Git is a modern and widely used version control system. We assume that you have already worked with another version control system and will solely focus on some Git basics and how this is can be used in the context of Saros. The central concept is a Git Repository, it is located in the .git/ subdirectory and holds the entire history of the project (branches, commits, all versions of files). The Git checkout command takes a specific revision from your local repository and updates, adds and removes files from your Git working directory to match the state of that revision. Git commit allows you to create a local commit that will be stored in your local Git Repository. The Git Index is a unique concept that represents the state that will be used for the next commit. You will mostly use it to add a new file, remove an existing one. An advanced usage of the Git Index is to partially add modifications of a file to the index, in the beginning you will probably not want to use that specific feature though. Git allows you to create local branches that are branched off any Git commit that is already in the local Git Repository. In case you created a Git Commit too quickly you can always amend to the last one. Once you want to share your changes you will need to Git push, this will move your local changes to the remote Git Repository. To get changes from a remote Git Repository you will need to use the Git pull. This will transfer all commits from the remote repository to your local one and it will try to move your changes to the top of the history. The last operation is called a Git rebase.

There are plenty of Git introductions available on the web, some describe the technical details, some are for people coming from Subversion, some show how to use the Eclipse integration. The Git website has a documentation section with links to good documentation. If the following guide is not enough you should be able to find information there.

4.1.2 Git clone with EGit (textual)

EGit Plugin versions
The below procedure was tested with Eclipse 3.7.1 and EGit plugin version 1.3.0.201202151440-r.
1. **BEGIN**
   
   File → Import

2. **SELECT IMPORT FROM GIT**
   
   Select Git → Projects from Git and press Next.

3. **SELECT REPOSITORY SOURCE**
   
   Select URI and press Next.

4. **SOURCE GIT REPOSITORY**
   
   Fill in `dpp.git.sourceforge.net` as Host, use `gitroot/dpp/saros` as repository path and select `git` as the protocol and finish by pressing the Next button.

5. **SELECT THE BRANCH**
   
   Select the `master` branch and continue by pressing Next.

6. **SELECT LOCAL DESTINATION**
   
   Decide where to store the cloned Git repository. The proposed directory should work and press Next.

7. **CLONING THE REPOSITORY**
   
   Progress of downloading the repository is shown.

8. **SELECT PROJECT DIRECTORY**
   
   Select `Import existing projects` and import the top level directory `Working Directory` and continue using the Next button.

9. **IMPORT PROJECTS**
   
   Select at least the `de.fu_berlin.inf.nebula` and `Saros` to import.

10. **SWITCH TO Git Perspective**
    
    Clicking on Window+Open Perspective → Other and select the **Git Perspective**.

11. **ENTER GERRIT CONFIGURATION**
    
    Unfold the `Saros`, `Remotes` and `origin` folder. Use the context menu and select **Gerrit Configuration**.

12. **CONFIGURE GERRIT**
    
    Use `ssh://saros-build.imp.fu-berlin.de:29418/saros.git` as the Push URI and `refs/for/master` as the Destination branch.

13. **VERIFY EGit IS ENABLED**
    
    Switch to the **Java Perspective** and use the context menu in the **Package Explorer** and verify that the Team contains options for commit.

### 4.1.3 Git clone with EGit (graphical)

1. **BEGIN**
2. **Select import from Git**

3. **Select Repository Source**

4. **Source Git Repository**
5. Select the branch

6. Select local destination

7. Cloning the repository

Host: dpp.git.sourceforge.net, Path: gitroot/dpp/saros
8. Select project directory

9. Import projects

10. Switch to the Git Perspective
11. **Switch to the Git Perspective**

12. **Enable Gerrit**

13. **Configure Gerrit**
I4. Verify EGit being used in the Java Perspective

4.2 Configuring Git for Gerrit

So far you managed to clone a remote Git repository and have checked out the default branch. To make local commits you will need to set a name and email address. If you do not do this Gerrit might reject your commits. If you want to make your changes available and ask for review you will need to push your local commits into the Gerrit Git repository. Git is using SSH to communicate with remote Git repositories, you will authenticate with your private SSH key. The following sections will show you how to do it.

4.2.1 Creating an SSH key

You can create SSH keys using Eclipse. The following procedure will explain the process.

1. Open the preferences
   Window → Preferences
2. **SELECT THE SSH CONFIGURATION**
   Enter SSH into the search field and select SSH2.

3. **SWITCH TO THE KEY MANAGEMENT TAB**

4. **GENERATE A RSA KEY**
   Select the Generate RSA Key to generate a new key pair. Optionally set a password. The selected text is the public key and needs to be used for Gerrit.
5. **Save the Key**

Press Save Private Key to store the new keypair. This will require clicking a couple of times.
4.2.2 Setting your name and email address

You will need to set your name and email address. Git has a very simple configuration format. You will need to set the keys `user.name` and `user.email` to the names of your choice. The below procedure will show you how to do it and then you are ready to make local commits.

1. **Open the Preferences**
   Window → Preferences

2. **Open the Git Configuration**
   Search for `Git` and select the Configuration page.
3. **ADD AN ENTRY FOR YOUR NAME**

   Press the New Entry button and use `user.name` as the key and set your name as the value.

4. **ADD AN ENTRY FOR YOUR EMAIL ADDRESS**

   Press the New Entry button and use `user.email` as the key and set your email address as the value.
4.3 Register on Gerrit

We are using Gerrit for code review and automatic integration of reviewed patches. You will need to create an account, pick a user name, register your email addresses and add your public SSH key. Gerrit will compare the committer email address in the Git commits with the registered email addresses for your SSH key.

4.3.1 Signing up using OpenID

Our Gerrit is doing authentication based on OpenID. This means you will need to have an account with an OpenID provider. Google and Yahoo are well known providers. After you signed up you need to pick a user name, add your email addresses and SSH keys. The first time you login all this can be done from the landing page.

1. Navigate to Gerrit

Go to http://saros-build.imp.fu-berlin.de/gerrit and press Sign In.

2. Select an OpenID Provider

Select your OpenID provider, e.g. Yahoo, Google, Livejournal, MyOpenID.

3. Sign into your OpenID Provider

You will need to sign into your account.
4. **AUTHORIZE THE USAGE BY GERRIT**
   Allow to use your identity by Gerrit.

5. **CONFIGURE THE GERRIT ACCOUNT**
   Your browser will return to our Gerrit and you will need to set your name, pick a user name, add your SSH keys. You should ask your supervisor to add you to the **Reviewer** group to be allowed to make reviews and submit code.
4.3.2 Concepts and Terminology

Change
Every time you push a commit with a new Change-Id Gerrit allocates a change. The change contains a number of patchsets, comments on the patchsets and ratings in various categories. Each change has a dedicated page that shows information about it. This includes dependencies between different changes, patchsets and the review comments.

Category
A category has a name and a rating scale. This is used to allow Users, Jenkins and other scripts to provide review. Right now we have the Verified, Review and Sanity categories are configured. The rating can range from -2 to +2.

Patchset
A patchset is a git commit that has been pushed to Gerrit. Gerrit uses the Change-Id of the commit message to identify the change. Each patchset can receive inline comments.

Submit
Once a change has received a +2 in the Review category and no negative voting in the other categories the last patchset can be submitted. This means Gerrit will now try to merge your patchset and mark the change as merged.

4.3.3 Making a review

Once you have opened the Gerrit page you can move to the list of changes that require review by clicking All. This will show you the list of all changes that require action. Clicking on any of these changes will bring you to the change overview. This page contains information about the change, already received voting, dependencies of the change with other changes and allows you to look at the current patchset and leave inline comments. Once you went through all the touched files you can click the Review button, leave a vote and provide a comment. The easiest way to start the review is by clicking on the Commit Message and then jump to the next file. Inside the Differences view you can modify the display by clicking on the Side-by-Side, Unified Preferences, Patch Sets links.

1. Switch to the All page

Select the change you want to review

2. Individual Change page
Check existing reviews, dependencies and scroll down to the patchsets and start your review by pressing on the Commit Message.

3. PROVIDE COMMENTS

Provide inline comments, or go to the Preferences to change the way the patch is displayed.

4. FINISHING THE REVIEW

Press the Up to Change to return to the change page and then the Review to finish your review and leave some final comments and rate the change.
4.3.4 Submitting a change

Once a patchset of a change has been approved the Submit Patchset will appear on the screen. Press the button to try to integrate your patch. The most common reasons that a patchset will not integrate is that a dependency is not integrated yet or the patchset can not be cleanly merged. For the later the following chapter will explain how you can rebase it and re-push it.

4.4 Summary

4.4.1 System Overview

In the previous sections we looked at the EGit and Gerrit integration and will now connect the dots between those systems. Once you have installed EGit you will need to Git clone the Saros repository, you can then make local changes, create branches, move files, etc. The local commits can be pushed from your local Git repository to the Gerrit system. The changes can now be reviewed in Gerrit by the Saros team. The Jenkins CI system will be triggered and will test your change and report back to Gerrit. Once the change is approved it can be submitted and will be merged into the official Git repository.
Clone a remote repository and checkout the default branch. This will create a local directory and populate the working directory.

Create local branches, add new files and make local commits. This will create local history in your Git repository.

Push your local branch to Gerrit for review. Gerrit will assign you a change number. The change can now be reviewed.

Jenkins will be notified about your change and will execute the test cases. The result of the tests will be reported back to Gerrit.

After a change is approved it can be submitted and will be integrated into the repository. Gerrit will push the changes to Github.

This will copy the current state of a remote repository into your local repository. Your current changes will move from the old last revision of the remote repository to the new one.

Figure 4.1: Code, Review and Test workflow
Chapter 5

Making changes to Saros

This chapter will show you how to work with EGit and Gerrit in the most common cases. This includes testing the code of someone else, making your first change, editing your change after having received review, updating your change to apply against the latest version of Saros. This will conclude with more advanced usage of having a chain of changes that build on top of each other and being asked to modify a change that is in the middle. This does not replace taking a look at the rather lengthy Eclipse EGit Wiki.

5.1 Cleaning your working copy

You might have created new classes and they are not under version version control yet and now you would like to switch the branch. Files not under version control will not be removed when switching branches. In some cases this might not be what you want and you could create a work in progress commit with the new files and later amend the commit with more changes and a better Git commit message. The other option is to remove files and this is what will be handled now.

5.1.1 Reverting local modifications

The easiest way to revert your un-committed modifications is to use the Git reset feature. You want to use the hard option. This will reset the last commit of your history to the selected commit and will modify your working directory to match this commit.

1. Enter the context menu and select Team → Reset.
2. Select the branch/last commit. The default is the last commit. Select hard as the reset type.
3. Confirm the reset option.
4. The package explores does not indicate any modification.

1. ENTER CONTEXTMENU
2. Select reset type

3. Confirm

4. Clean tree
5.1.2 Deleting untracked/new files

The *Git reset* feature does not remove files that are not under version control. The *Team Synchronizing* perspective will show you files not under version control. You can easily delete these files in this perspective.

1. Use the Contextmenu and then Team → Synchronize Workspace.
2. Confirm moving to the *Team Synchronizing* perspective.
3. Select the files you want to remove and use the Contextmenu to delete them.

1. ENTER CONTEXTMENU

2. Switch perspective
3. **DELETE FILES**

4. **CONFIRM**

5. **NO CHANGES LEFT**
5.2 Downloading a change

A member of your team has made some changes and uploaded them to Gerrit. You might want to test his change and the EGit plugin has support to make that very easy. The below will go through the necessary steps to fetch a change from Gerrit into a new local branch. The procedure assumes that you are in the *Git Perspective*.

### 5.2.1 Downloading a change (textual)

1. **Select Fetch From Gerrit**
   Enter the context menu on the *Git Repository* and select the Fetch From Gerrit.

2. **Select a change by CTRL+I**
   You can enter the change directly or use CTRL+I to fetch a list of available changes and select it from a dropdown menu or enter it directly.

3. **Select a change from the dropdown**
   Select the change and the patchset you want. If you enter it directly use e.g. `refs/changes/47/47/3`, the format is to give the last two digits of the change, followed by the complete change number and then select the patchset you want.

4. **Create a local branch**
   Pick a local name and put it into the *Branch name* lineedit. This way you will have a local branch with the chosen name.

5. **Switched to local branch**
   EGit will indicate that it switched to your new branch.

### 5.2.2 Downloading a change (graphical)

1. **Select Fetch From Gerrit**
2. **Select a change by CTRL+I**

3. **Select a change from the dropdown**

4. **Create a local branch**
5. Switched to local branch

5.3 Doing your first change

Hopefully sooner than later you are ready to send your first contribution. You will need to create a Git commit and post it for review to our Gerrit installation. General information about using EGit can be found in the Eclipse EGit Wiki.

5.3.1 The first change (textual)

1. Open any file
   E.g. open `Utils.java` file.

2. Make a modification
   E.g. remove the `getFreePort` method.

3. Add change to the index
   Use the context menu of the file to access Team → Add to Index. This moves the change into the index and marks the new version as to be committed.

4. Commit
   Commit all changes that are in the index.
5. **Writing a Commit Message**
   Make sure that the Gerrit option is enabled and that a line with `Change-Id` is in the editor field. Follow our [Commit Guidelines](#) for writing the message.

6. **Switch to the Repository View**
   Use the context menu and Team → Show in Repository View to switch to another view to be able to push your changes.

7. **Push**
   Use the context menu on the repository and select the Push item.

8. **Select Target**
   Select the pre-configured Gerrit repository as the target for your changes and continue by pressing the Next button.

9. **Select Branch**
   Select what and where to push things. The default Destination Ref should be `refs/for/master` and press Next to continue.

10. **Push Confirmation**
    Review what and where things are pushed and Finish.

11. **Push Result**
    The result should include a set of change numbers of the Gerrit system.

### 5.3.2 The first change (graphical)

1. **Open a Util.java**

2. **Make a Modification**
3. **ADD CHANGE TO THE INDEX**

4. **COMMIT**

5. **WRITING A COMMIT MESSAGE**
6. **Switch to the repository view**

7. **Push**

8. **Select Target**
9. **Select Branch**

10. **Push Confirmation**

11. **Push Result**
5.3.3 Understanding build failures

After you pushed your changes to Gerrit, the Saros-Gerrit job will be scheduled on the Jenkins CI system. The execution of this job might fail and the most common errors are missing files, build failures due building on an older OSGi/Eclipse version or test failures because the tests are run without SWT/GUI being initialized. The below will help you to identify the reason of the failure.

In case the build has failed it can be difficult to find the error in the ant log. The easiest is to search for `.ERROR` in the console log of the job. An example for a build failure is Job 24, try to find the error.

5.3.4 Dealing with feedback

Imagine you made three commits on top of each other and have pushed them to Gerrit. Now either you or a reviewer has found something that should be modified in any of the separate changes. The straightforward way is to use the built-in Gerrit support to create a new branch with these changes, make the desired modifications, `amend` to the old commit and push it to Gerrit again.

1. **Fetch from Gerrit**
   
   From within the Git repository view select the context menu and press Fetch from Gerrit

2. **Select a Change**
   
   You will need to select a change or use Control+Space to search for changes.
3. **Pick a Local Branch Name**

Select a name for the local branch.

4. **Start Your Modification**

You are on a branch that contains your previous change. You are now able to do the desired modification.

5. **Amending Your Commit**
6. Pushing your commits

If you push your current branch to Gerrit again, Gerrit will update the patches in the changes.

7. Deleting a branch

You can now checkout a different branch and delete the old one.

5.4 Updating your changes

5.4.1 Fetch, Rebase and Push

You have started your work, created a local branch based on the last commit of origin/master. During the time you made your changes and now some other changes were integrated into origin/master and you want to update. This can be done by combining two operations. The first is Git fetch to fetch all changes from the origin into your local Git repository. This operation will update the origin/master branch to the state of the remote Git repository. The second operation is Git rebase. This will move any local commits to the new origin/master. The operation is called Git rebase because your local changes will be moved from the old state of origin/master to the current state of it. During the Git rebase local commits can vanish as they are already included, or you can end with a conflict. In that case you will be presented with multiple options, to abort the rebase, to skip (remove) the commit that caused the conflict or to attempt to merge it using the mergetool. Once you have resolved the conflicts use Team → Rebase → Continue to continue with the next local Git commit.
1. Enter the Contextmenu, select Team → Fetch from Upstream.
2. An overview with changes appear
3. Enter the Contextmenu, select Team → Rebase.
4. Select the branch you want to rebase against.
5. On conflict a dialog will ask you to make a decision on how to resolve it. Once it is resolved select Team → Rebase → Continue in the contextmenu to continue with the next commit.
6. Your branch is now based on the new origin/master

1. Fetch from Upstream

2. Result

3. Begin Rebase
4. **Select Branch**

5. **Conflict**

6. **Abort**
When the rebase goes wrong

When the rebase goes wrong you have two options. You can either select Team → Rebase → Abort or if you notice after the rebase the Git Reflog contains your changes before the Git rebase and you can checkout this version again.

External Git mergetool

On OSX and Linux you can use git mergetool on the command line to use opendiff/kdiff3/meld to the merging.

5.5 Advanced Git Topics

You might have started to use Git beyond the featureset of SVN and have a chain of local commits. You have asked to review your chain of changes and a peer has found an issue in of the changes in the middle of the chain. From the previous chapters you have seen how to the modification of a single change and this is what you need to do first, the next thing is to rebase the commits that followed the original one. And below there will be two alternatives for doing that.

5.5.1 Cherry-pick

TODO: describe cherry-pick

5.5.2 Staging Changes

TODO

5.6 Tasks during a release

For the release two additional tasks are required. The first is to create the branch and the second is to push bugfixes to the right branch. In both cases this is done by selecting another branch in the destination branch when pushing your changes. In case of creating the release branch the right destination branch is refs/heads/release/DATE, asking for review is done through the refs/for/release/DATE branch.
Task | Destination Reference
--- | ---
Create Release Branch | refs/heads/release/DATE
Change for Release | refs/for/release/DATE

Table 5.1: Destination Reference

5.6.1 Pushing to a non default branch (textual)

1. **SELECT PUSH IN THE CONTEXT MENU**
   Inside the Git repository view activate the context menu and select the Push item.

2. **SELECT THE CONFIGURED REMOTE REPOSITORY**
   The configured remote repository is selected by default, move to the next page by pressing the Next button.

3. **SPECIFY THE BRANCH NAME**
   Inside the Specifications for push area change refs/for/master to the branch name mentioned above and continue to the next page.

4. **PUSH CONFIRMATION**
   Confirm the push and finish the dialog

5.6.2 Pushing to a non default branch (graphical)

1. **SELECT PUSH IN THE CONTEXT MENU**

2. **SELECT THE CONFIGURED REMOTE REPOSITORY**
3. **Specify the branch name**

4. **Push confirmation**
   
   Confirm the push and finish the dialog.
Chapter 6

Patterns and Anti-Patterns

There are various patterns we want to be used and there have been various anti-patterns we don’t want to see again. This section provides an illustration of what is good and what should be avoided and give reasons about it.

Under Construction
Not completed yet.

6.1 Anti-Patterns

6.1.1 PicoContainer usage

The PicoContainer will resolve dependencies between objects. For this to work properly one needs to have all dependencies as parameters of the constructors. One common misuse is to get a dependency with an indirection.

```java
public class SkypeManager {
    private final SarosNet sarosNet;
    private final IPreferenceStore preferences;

    public SkypeManager(Saros saros) {
        this.sarosNet = saros.getSarosNet();
        this.preferences = saros.getPreferenceStore();
    }
}
```

The above code has dependencies on SarosNet and IPreferenceStore and they are resolved through the Saros instance. This is a problem when it comes to testing, there might not be a mock for the SarosNet or the IPreferenceStore store and one might have an NullPointerException. The below code shows how to properly express the dependencies.

```java
public class SkypeManager {
    private final SarosNet sarosNet;
    private final IPreferenceStore preferences;

    public SkypeManager(SarosNet sarosNet, IPreferenceStore preferences) {
        this.sarosNet = sarosNet;
        this.preferences = preferences;
    }
}
```
6.1.2 Use Interfaces

TODO: E.g. ISarosSession vs. Session, talk about testability

6.1.3 Splitting Work

Describe.. splitting work across different threads and synch. issues

6.1.4 ...

MORE
Chapter 7

Configuration of Gerrit

This is mostly a guide for the project owner. Our Gerrit has some project specific configuration that is explained in this chapter.

7.1 Grant push access to Gerrit

Users in the Approved group are allowed to push. Add new users to this group.

7.2 Granting +2/-2 and the right to submit

Users in the Approvers group are allowed to set +2/-2 in the review category, override the verified ranking and are allowed to submit approved changes.

7.3 Granting branch creation

Users in the Release-Managers group are allowed to push merges for review and to create new branches.