16.2. Advantages of code review 49
17. The Gerrit code review system 51
17.1. What is Gerrit? 51
17.2. Change 51
17.3. How does Gerrit work? 51
17.4. Review categories - Verified and Code-Review 52
17.5. Voting in Gerrit 52
17.6. Gerrit online documentation 52
17.7. Posting draft reviews 53
18. Gerrit workflow description 55
18.1. A typical Gerrit workflow 55
18.2. Gerrit vs. Git workflow 55
18.3. Gerrit vs. GitHub workflow 55
19. Working with Gerrit reviews 57
19.1. Creating and updating a Gerrit change request from Eclipse 57
19.2. Gerrit support in Eclipse 59
19.3. URL for reviews 60
19.4. Reviewing a change request 60
19.5. Publishing your review result 61
19.6. Improve a change based on review feedback 62
19.7. Keyboard shortcuts in Gerrit 63
19.8. Fetching a remote change into a local Git repository 63
19.9. Fetching a remote change with Eclipse Git 64
19.10. Edit a change directly via the web interface 64
20. Solving technical Gerrit problems and general watchouts for contributions 67
20.1. You forgot to amend the commit 67
20.2. Handling merge conflicts 68
20.3. non-fast forward 68
20.4. Remove a bad commit from a series of commits 69
20.5. More error messages and their solution 70
20.6. Things to consider if you want to contribute to an existing project 70
20.6.1. How to start contributing? 70
20.6.2. Scope of a Gerrit change / commit 71
20.6.3. Commit message 71
20.6.4. Copyright header update 72
20.6.5. Example: Copyright header in Eclipse Platform UI 72
20.6.6. Stay polite 73
20.6.7. Avoid unnecessary formatting changes 73
20.6.8. Avoid unnecessary whitespace changes 73
20.6.9. Pushing unfinished features 74
20.6.10. Valid user and email 74
20.6.11. Not getting feedback 74
20.6.12. Dealing with negative feedback 74
21. Reviewing changes 75
21.1. Doing a review 75
21.2. Seeing your reviews 75

IV. Introduction to unit testing and the Eclipse test suites 77

22. Unit tests, integration tests and performance tests 79
    22.1. What are software tests? 79
    22.2. Why are software tests helpful? 79
23. Testing terminology 81
    23.1. Code (or application) under test 81
    23.2. Test fixture 81
    23.3. Unit tests and unit testing 81
    23.4. Integration tests 81
    23.5. Performance tests 81
    23.6. Behavior vs. state testing 82
24. Unit testing with JUnit 83
    24.1. The JUnit framework 83
    24.2. How to define a test in JUnit? 83
    24.3. Example JUnit test 83
    24.4. JUnit naming conventions 84
    24.5. JUnit naming conventions for Maven 84
    24.6. JUnit test suites 84
    24.7. Run your test from the command line 84
25. Basic JUnit code constructs 87
    25.1. Available JUnit annotations 87
    25.2. Assert statements 87
    25.3. Test execution order 88
26. Eclipse and tests 89
    26.1. Eclipse projects and their test suite 89
    26.2. Version of JUnit 89
    26.3. Tips: running the unit tests on a virtual server 89
27. Running the Eclipse platform unit tests 91
    27.1. Repositories for platform tests 91
    27.2. Example: JFace tests 91

V. Introduction to Eclipse plug-in development 93

28. Installation for extending the Eclipse IDE 95
    28.1. Java requirements of the Eclipse IDE 95
    28.2. Download the Eclipse SDK 95
    28.3. Installation the Eclipse IDE 96
        28.3.1. Install the Eclipse IDE 96
        28.3.2. Solving exit code=13 while starting the Eclipse IDE 96
29. Architecture of Eclipse plug-ins 97
    29.1. Architecture of Eclipse based applications 97
    29.2. Core components of the Eclipse platform 98
    29.3. Eclipse API and internal API 98
    29.4. Important configuration files for Eclipse plug-ins 100
30. Starting an Eclipse instance (Runtime Eclipse) 101
    30.1. Starting the Eclipse IDE from Eclipse 101

30.2. Starting a new Eclipse instance 101
30.3. Debugging the Eclipse instance 102
31. The usage of run configurations 103
31.1. What are run configurations? 103
31.2. Reviewing run configurations 103
31.3. Run arguments 104
32. Adding e4 commands, menus and toolbars to a 3.x API based applications 107
32.1. Adding e4 menu entries 107
32.2. Error analysis 109
32.3. Adding e4 toolbar entries to the appliation window 109
33. Exercise: Add a e4 menu and toolbar to the Eclipse IDE 111
33.1. Target of this exercise 111
33.2. Creating a plug-in project 111
33.3. Starting an Eclipse runtime IDE with your plug-in 115
33.4. Adding the plug-in dependencies for the e4 API 115
33.5. Creating the handler class 115
33.6. Creating a model contribution 116
33.7. Adding a toolbar contribution 118
33.8. Validating the presence of the menu and toolbar contribution 120
34. Features and feature projects 121
34.1. What are feature projects and features? 121
34.2. Creating a feature 121
34.3. The purpose of the tabs in the feature editor 121
34.4. Advantages of using features 122
35. Deploy your software component (plug-ins) locally 123
35.1. Options for installing a plug-in 123
35.2. Installing your plug-in from your Eclipse IDE 123
35.3. Export plug-in and put into dropins folder 124
35.4. Deployment using an update site 125
36. Exercise: Create a feature for your plug-in 127
36.1. Create a feature project 127
36.2. Create a category definition 128
37. Exercise: Create update site and install plug-in from it 131
37.1. Create an update site 131
37.2. Install feature via the Eclipse update manager 133
37.3. Validate installation 135
38. Patching existing Eclipse plug-ins with feature patches 137
38.1. Feature patch projects 137
38.2. Feature patch projects 137

VI. Tools for accessing the Eclipse source code 139

39. Eclipse spy functionality for finding things 141
39.1. Plug-in Spy for UI parts 141
39.2. Menu spy 141
39.3. SWT Spy 142

40. Using the e4 spies 143
40.1. The e4 tools project 143
40.2. How to install the e4 tools 143
40.2.1. Install the e4 tools 143
40.2.2. Install the e4 spies from the vogella GmbH 144
40.2.3. Install the e4 spies from Eclipse.org 144
40.3. Analyzing the application model with the model spy 145
40.4. Model spy and the Eclipse IDE 146
40.5. CSS Tools 146
40.5.1. CSS Spy 146
40.5.2. CSS Scratchpad 147
41. Using the search functionality 149
41.1. Filtering by the Java tools 149
41.2. Include all plug-ins in Java search 149
41.3. Find the plug-in for a certain class 149
41.4. Finding classes and plug-ins 150
41.4.1. Open Plug-in artifact 150
41.4.2. Plug-in search 151
41.4.3. Plain text search 152
41.5. Example: tracing for key bindings 153
42. Tracing 155
42.1. What is tracing? 155
42.2. Turning on tracing via an options file 155
42.3. Turning on tracing at runtime 156
43. Updating the copyright header of a source file 159
43.1. Why updating the copyright header of a changed file? 159
43.2. Using the Eclipse releng tooling for updating the copyright header 159

VII. Building your custom IDE distribution 161

44. Creating a custom Eclipse IDE build 163
44.1. Building the Eclipse IDE 163
44.2. Requirements 163
44.3. Cloning the SDK repository 163
44.4. Building the Eclipse IDE 163
44.5. Cleanup before the next build 164
45. Additional information about building the Eclipse platform 165
45.1. Reporting issues or asking questions about the build 165
45.2. Eclipse platform Hudson builds 165
45.3. Release and milestone builds 165
45.4. Changing build ID 166
45.5. Build single parts of the aggregator 166
45.6. Building natives (SWT binary files) 166
45.7. Fedora Eclipse CBI build 166

VIII. Eclipse foundation staff and Eclipse project lead interviews 167

46. Mike Milinkovich about the vision of Eclipse 169
46.1. Who are you and what is your role in the Eclipse organization? 169
46.2. Where do you see the cornerstones of the Eclipse OS project? 169
46.3. What is your vision for the future of the Eclipse OS project? 170
47. Wayne Beaton and Ian Skerrett about the Eclipse community work 171
47.1. Who are you and what is your role in the Eclipse organization? 171
47.2. Why should someone consider contributing to Eclipse? 171
47.3. How does the Eclipse foundation helps the Eclipse projects? 172
47.4. Which tools do you provide which are most valuable for developers? 172
47.5. What improvements can contributors and committers expect? 172
47.6. What role does marketing play in the Eclipse foundation? 173
47.7. If someone wants to do a community event, how should he get into contact with the foundation? 173
48. Denis Roy about the Eclipse infrastructure 175
48.1. Who are you and what is your role in the Eclipse organization:? 175
48.2. Why should someone be interested in contributing to Eclipse? 175
48.3. Which tools are the most valuable for committers? 175
48.4. Which tools are most valuable for contributors? 175
49. The Java development tools (JDT) project 177
49.1. Can you describe the target and scope of the JDT project? 177
49.2. Who are you and how are you involved in the Eclipse community? 177
49.3. Can you describe the project team and culture? 177
49.4. Where can a contributor find information how to contribute? 177
50. The Eclipse Git project 179
50.1. Can you describe the target and scope of the EGit project? 179
50.2. Who are you and how are you involved in the Eclipse community? 179
50.3. Can you describe the project team and culture? 179
50.4. Where can a contributor find information how to contribute? 180
51. The m2e project for Eclipse Maven integration 181
51.1. Can you describe the target and scope of the m2e project? 181
51.2. Who are you and how are you involved in the Eclipse community? 181
51.3. Can you describe the project team and culture? 181
51.4. Where can a potential contributor find information about potential contributions? 181
52. The C and C++ IDE (CDT) project 183
52.1. Can you describe the target and scope of the CDT project? 183
52.2. Who are you and how are you involved in the Eclipse community? 183
52.3. Can you describe the project team and culture? 184
52.4. Where can a contributor find information how to contribute? 184
53. The Tycho project 187
53.1. Can you describe the target and scope of the Tycho project? 187
53.2. Who are you and how are you involved in the Eclipse community? 187
53.3. Can you describe the project team and culture? 187
53.4. Where can a contributor find information how to contribute? 188
54. The Standard Widget Toolkit (SWT) project 189
54.1. Can you describe the target and scope of the SWT project? 189

54.2. Who are you and how are you involved in the Eclipse community? 189
54.3. Can you describe the project team and culture? 189
54.4. Where can a contributor find information how to contribute? 190

A. Additional Eclipse resources 191

A.1. Eclipse online resources 191
A.1.1. Online documentations 191
A.1.2. Web resources 191
A.2. Eclipse Rich Client Platform (RCP) 191
A.3. More books from the vogella book series 192

B. Additional information 193

B.1. Installation of Git support into Eclipse 193
B.2. How to configure the usage of Git in Eclipse 193
B.2.1. Interoperability of Git command line settings with the Eclipse IDE 193
B.2.2. Git user settings in Eclipse 194
B.2.3. Default clone location 194
B.2.4. Configuring the toolbar and the menu for Git usage 194
B.3. Git user configuration for the Eclipse IDE 196
B.3.1. Validate your Git user settings 196
B.3.2. Configure Git to rebase during pull operations 196
B.4. Authentication via SSH 197
B.4.1. The concept of SSH 197
B.4.2. SSH key pair generation 197
B.4.3. Eclipse support for SSH based authentication 199
B.5. Appendix: Cloning from the Git server and adjusting the push URL 200
B.5.1. Overview 200
B.5.2. Gerrit push configuration 201
B.6. Using the Eclipse installer for code contributions 203

Index 209

Once you have a user account, you can login to the Eclipse bug tracker. This allows you to comment on existing bugs and report new ones. The user data for the all Eclipse sites are the same, i.e, the forum, marketplace, bug tracker, etc. Only for the Gerrit access, different user data is used.

As example you can report bugs for the Eclipse platform via the following link: *Bug report for the Eclipse platform [https://bugs.eclipse.org/bugs/enter_bug.cgi?product=Platform]*.

9.3. Eclipse bug priorities

The Eclipse Bugzilla system allows you and the Eclipse committer to enter the bug priority. But overall, it is up to each project do decide how they handle bugs so some variation from project to project will occur. The following rules can be used as guideline.

Table 9.1. Bug priorities

Priority	Description
blocker	The bug blocks development or testing of the build and no workaround is known.
critical	Implies "loss of data" or frequent crashes or a severe memory leak.
major	Implies a "major loss of function".
normal	This is the default value for new bug reports. Implies some loss of functionality under specific circumstances, typically the correct setting unless one of the other levels fit.
minor	Something is wrong, but doesn't affect function significantly or other problem where easy workaround is present.
trivial	This describes a cosmetic problem like misspelled words or misaligned text, but doesn't affect function.
enhancement	Represents a request for enhancement (also for "major" features that would be really nice to have).

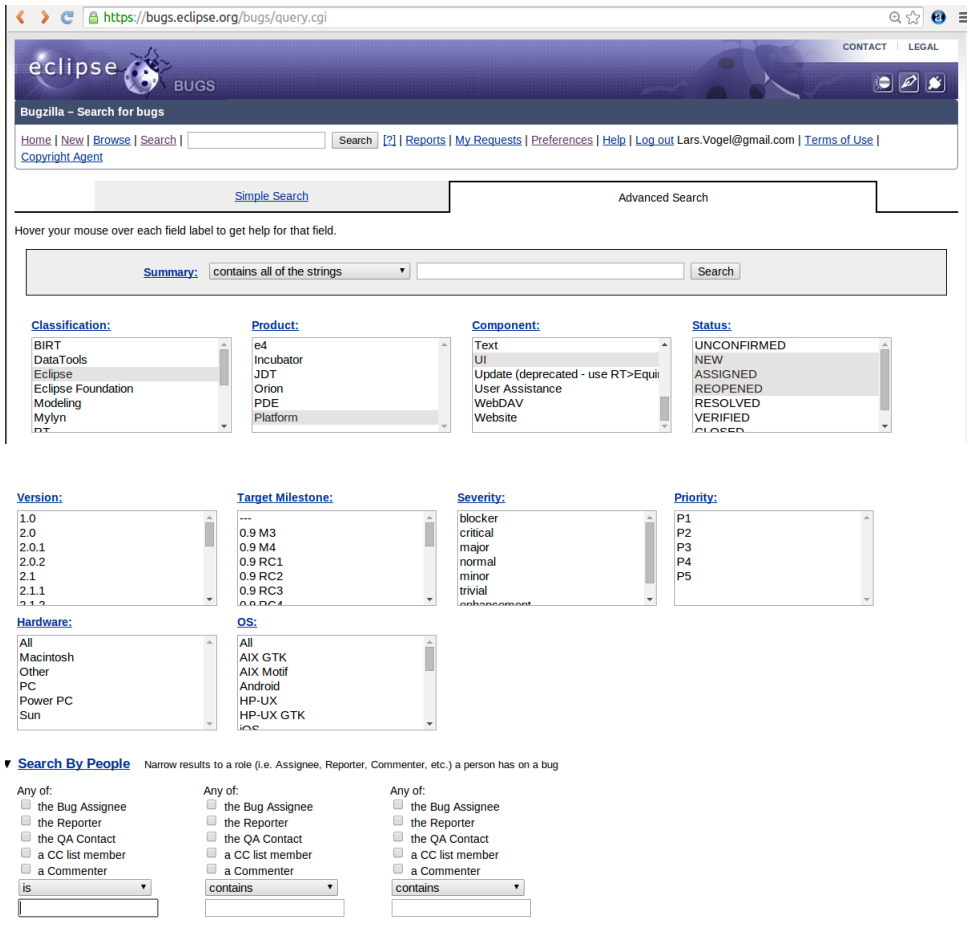
9.4. Target

In this exercise you use the Bugzilla system to review some of the Eclipse platform bugs. No action is excepted from you, but if you find an updated bug, you should update the bug report and describe that the problem is solved.

This exercise uses the Eclipse platform as example but you can use any Eclipse project of your choice.

9.5. Run Bugzilla query

Open to *Eclipse Bugzilla [https://bugs.eclipse.org/bugs/]* and select the *Search* button. Select the *Advanced Search* tab and search for *Eclipse* → *Platform* → *UI* for all bugs in status *NEW*, *ASSIGNED*, *UNCONFIRMED* and *REOPENED*.



In most cases Eclipse project have tons of unsolved bugs. If you are looking for existing software bugs, it is recommended to look at the latest bugs, e.g., the bugs which have been recently updated.

10
Eclipse projects responsible for the Eclipse Java IDE

10.1. The Eclipse project

The Eclipse IDE is an open source (OS) project. Eclipse.org hosts lots of OS projects with different purposes, but the foundation of the whole Eclipse IDE is delivered by the Eclipse platform, the Java development tools (JDT) and the Plug-in Development Environment (PDE) project.

All these projects belong to the project called "Eclipse project". The name is a bit strange since nowadays there are many Eclipse projects. But it originates from the fact that these projects were at the beginning the only available Eclipse projects.

These components are bundled together as the Eclipse Standard Development Kit (SDK) release. This is also known as the Eclipse standard distribution. The Eclipse SDK contains all tools to develop Eclipse plug-ins.

In addition to these projects, the platform also has an incubator project called *e4* which provides new tools for Eclipse 4 developments. It also serves as a testing ground for new ideas.

10.2. The Eclipse platform projects home pages and Git repositories

The following table lists the home pages of these projects along with the developer pages, which list the Git (and Gerrit) repositories.

Table 10.1. Eclipse projects

Project	Home page	Developer pages with Git repositories
Platform	*Eclipse platform project [http://www.eclipse.org/eclipse/]*	Not relevant, as platform is only the top project
Platform UI	*Eclipse platform UI project [https://www.eclipse.org/eclipse/platform-ui/]*	*Platform UI developer page [https://projects.eclipse.org/projects/eclipse.platform.ui/developer]*
Java development tools (JDT)	*JDT project [http://www.eclipse.org/jdt]*	*JDT UI developer page [https://projects.eclipse.org/projects/eclipse.jdt.ui/developer]* and *JDT core developer page [https://projects.eclipse.org/projects/eclipse.jdt.core/developer]*
Plug-in Development Environment (PDE)	*PDE project [http://www.eclipse.org/pde/]*	*PDE developer page [https://projects.eclipse.org/projects/eclipse.pde/developer]*
e4	*e4 project [http://www.eclipse.org/e4/]*	*e4 developer page [https://projects.eclipse.org/projects/eclipse.e4/developer]*

10.3. Who is involved?

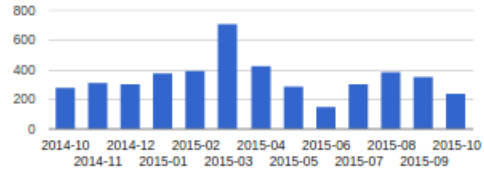
Despite the general assumption that the Eclipse SDK components are developed by an army of developers, the actual core developer team is relatively small.

General information about an Eclipse project can be found under *http://projects.eclipse.org/*. For example via the *Who's Involved* link you can see an overview of the project activities. For example, the information for the platform project can be found under the *Eclipse project [https://projects.eclipse.org/projects/eclipse/who]* link.

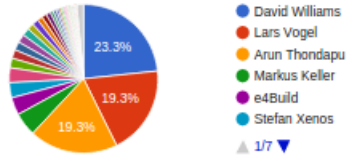
Eclipse Platform

Overview | Downloads | **Who's Involved** | Developer Resources | Governance | Contact Us

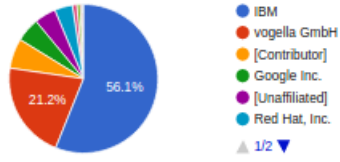
Contribution Activity:
Commits on this project (last 12 months).



Individual Contribution Activity:
Commits on this project by individuals over the last three months.



Organization Contribution Activity:
Commits on this project by supporting organization over the last three months.



Active Member Companies:
Member companies supporting this project over the last three months.

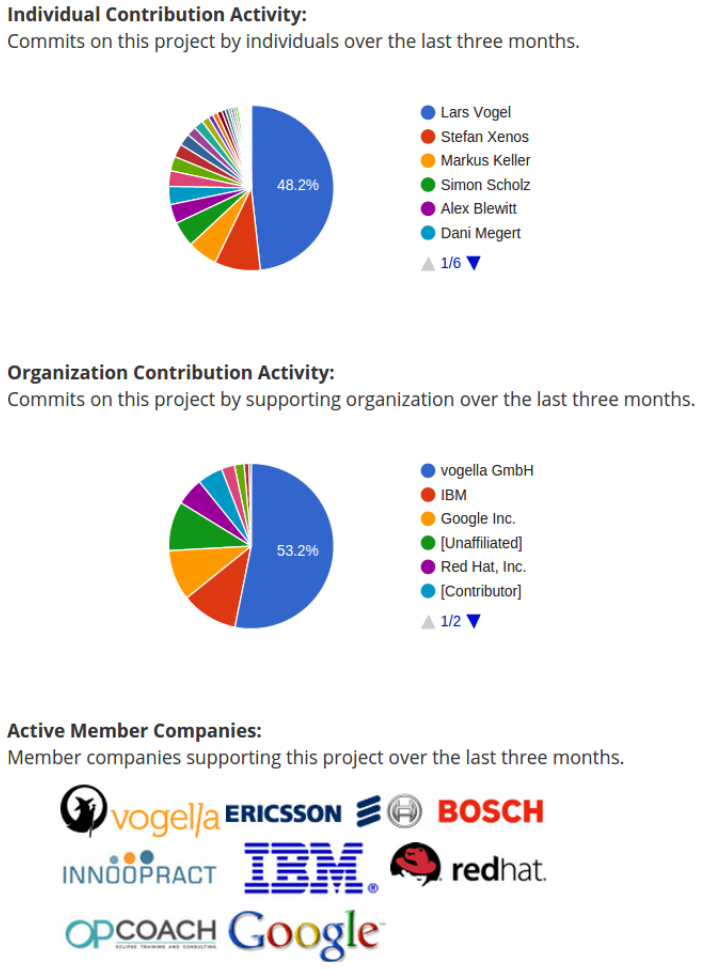
INNOOPRACT intel eclipse FOUNDATION MEMBER Google ERICSSON redhat.
IBM vogella OPCOACH BOSCH

Tip

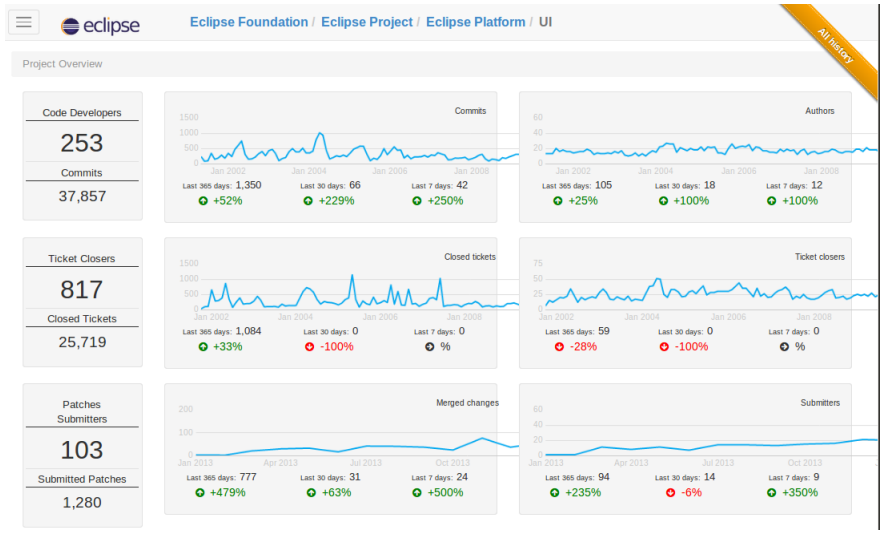
To find out how many people contribute to a given Eclipse Git repo you can use the following command on a Linux like system. For example running this command shows 27 on July, 07, 2014 result in 27 people contributing to the platform.ui repository.

```
git log --since='last 3 month' --raw | grep "^Author: " | sort | uniq -c | wc -l
```

Since the core team of an individual project is relatively small, contributors can really make a difference. For example the following screenshot shows a snapshot of the activities of the *Eclipse platform.ui project [https://projects.eclipse.org/projects/eclipse.platform.ui/who]*.



An alternative view gives the new dashboard. For example the following screeenshot shows the *Platform.ui dashboard [http://dashboard.eclipse.org/project.html?project=eclipse.platform.ui]* link.


eclipse
Eclipse Foundation / Eclipse Project / Eclipse Platform / UI
All history
Project Overview
Code Developers
253
Commits
37,857
Commits
Last 365 days: 1,350 +52%
Last 30 days: 66 +229%
Last 7 days: 42 +250%
Authors
Last 365 days: 105 +25%
Last 30 days: 18 +100%
Last 7 days: 12 +100%
Ticket Closers
817
Closed Tickets
25,719
Closed tickets
Last 365 days: 1,084 +33%
Last 30 days: 0 -100%
Last 7 days: 0 %
Ticket closers
Last 365 days: 59 -28%
Last 30 days: 0 -100%
Last 7 days: 0 %
Patches Submitters
103
Submitted Patches
1,280
Merged changes
Last 365 days: 777 +479%
Last 30 days: 31 +63%
Last 7 days: 24 +500%
Submitters
Last 365 days: 94 +235%
Last 30 days: 14 -6%
Last 7 days: 9 +350%

Part II. Preparing code contributions to Eclipse platform

11
Eclipse user creation and Gerrit server configuration

11.1. Create an Eclipse user account

To contribute to an Eclipse project you need to have valid user credentials. If you already have an Eclipse Bugzilla or Eclipse Forum account, you have already done this step. These systems use the same account. You can create a new user via the following: *Create Eclipse.org user account [https://dev.eclipse.org/site_login/createaccount.php]*.

11.2. Sign the Contributor License Agreement (CLA)

You also need to confirm to the Eclipse foundation that you have the right to contribute your code to the Eclipse open source project. This requires that you sign a contributor license agreement (CLA) via a web interface. Signing is really simple and only takes a few minutes.

The Eclipse CLA FAQ describes the process and purpose of the CLA. At the end this document contains a link for signing the CLA. You find it under the following URL: *Eclipse CLA FAQ [http://www.eclipse.org/legal/clafaq.php]*.

Tip

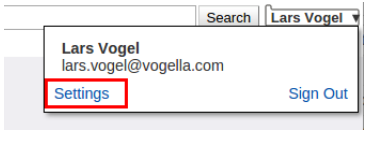
In case you contribute to the Eclipse open source project during your work time, ensure that your employer agrees that the code can be contributed.

11.3. Configure your Eclipse Gerrit user

You need to configure Gerrit if you want to use SSH or HTTPS to upload your contributions. The Gerrit user is NOT the same as your Eclipse user. Also, the Gerrit HTTPS password is NOT the same as your Eclipse user password.

To configure your access method open the following URL: *Gerrit at Eclipse [https://git.eclipse.org/r/]* and login with your Eclipse account (registered email address) and password via the top right corner of your browser.

Afterwards select *Settings* as highlighted in the following screenshot.

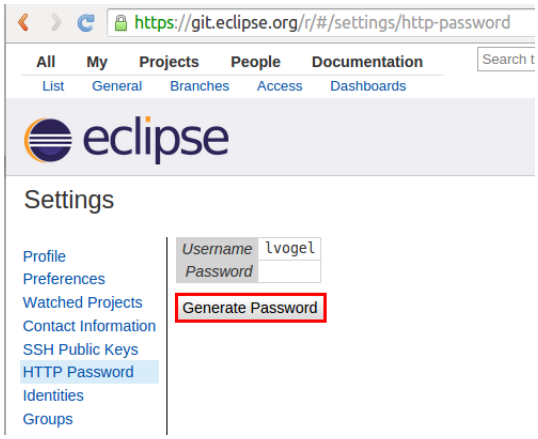


You can either use SSH or HTTPS to push to Gerrit. See Section 11.4, “Authentication via HTTPS” for the configuration of HTTPS and Section 11.5, “Authentication via SSH” for the configuration of the SSH access. If you are not familiar with SSH, the HTTPS authentication is most likely easier to

configure for you. If your company is using a proxy, you may have to use HTTPS as some proxies block SSH.

11.4. Authentication via HTTPS

The web page of the Gerrit review system enables you to generate an HTTP password. Select the *HTTP password* in the Gerrit user settings and generate a password. This setting is depicted in the following screenshot (the password is obfuscated).

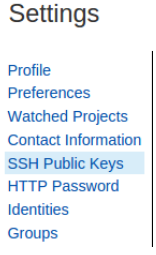


Take note or copy the password, as it will be needed to push your changes to the Eclipse Gerrit system. In contrast to SSH, when using HTTPS you will have to enter your password for each operation with Gerrit. You can visit this page later if you forgot the password.

11.5. Authentication via SSH

You need to upload your SSH key to the Gerrit installation so that you can push changes to Gerrit via SSH.

Once you have created a SSH key pair, upload your public SSH key to Gerrit to be able to push to it.

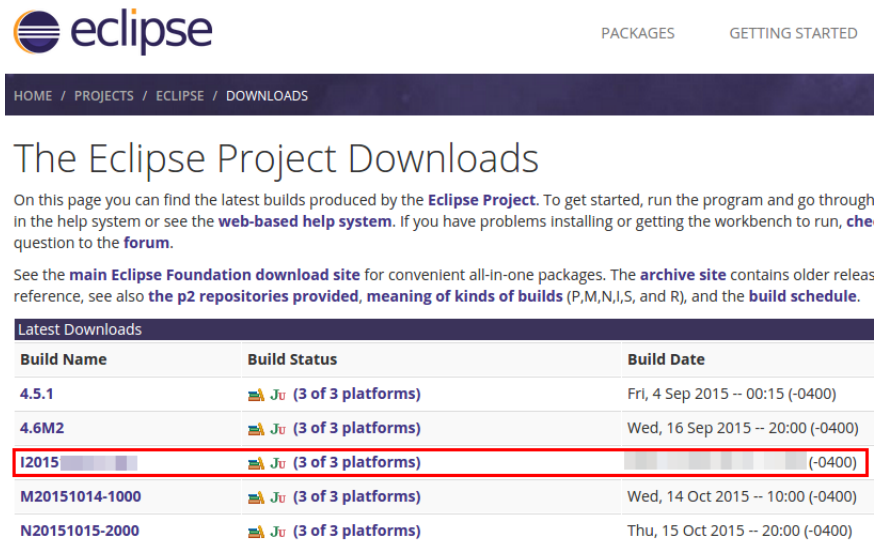


See Section B.4, “Authentication via SSH” more information on SSH Section B.4.3, “Eclipse support for SSH based authentication” for instructions how to create your SSH key via the Eclipse IDE.

12
Installation

12.1. Download the latest integration build

You can download the latest integration build for the next Eclipse release under the following URL: *Eclipse developer builds [http://download.eclipse.org/eclipse/downloads/]*.



This is sometimes necessary to test the latest features or to test a build which is later promoted to M or RC build by the Eclipse platform team.

> **Note**
>
> For several Eclipse projects you can also use the Eclipse installer. See Section B.6, “Using the Eclipse installer for code contributions” for a small description. At the time of this writing the installer does not work well for platform contributions hence is not the preferred way for contributing to it.

12.2. Install the Git tooling into the Eclipse IDE

After you downloaded a recent build for the Eclipse IDE, you need to install a few more plug-ins, most notable the Eclipse Git tooling.

12.3. Installation of Git

See Section B.1, “Installation of Git support into Eclipse” for installation instructions of Git.

12.4. Eclipse setup and user configuration

See Section B.2, “How to configure the usage of Git in Eclipse” and Section B.3, “Git user configuration for the Eclipse IDE” for installation instructions of Git.

Note

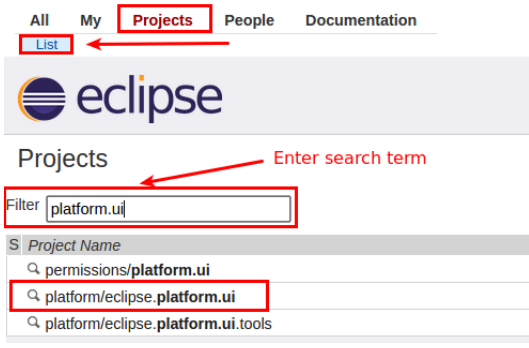
To contribute to Eclipse via Gerrit, your user and email must be the same as the Eclipse user credentials from Section 11.1, “Create an Eclipse user account”.

13
Clone the Eclipse platform.ui repository

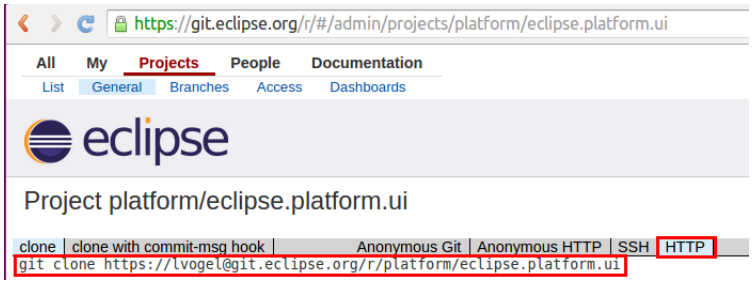
13.1. Finding the correct Git repository

Ensure you are logged into the Gerrit server webpage. This will enable you to use SSH or HTTPS for cloning the repository.

Go to the *Gerrit Eclipse homepage [https://git.eclipse.org/r/]* and select the *Projects → List* menu entry. This opens the *Project search page [https://git.eclipse.org/r/#/admin/projects/]*. Afterwards search for the project you want to contribute to. In our example we want to contribute to the platform.ui project.

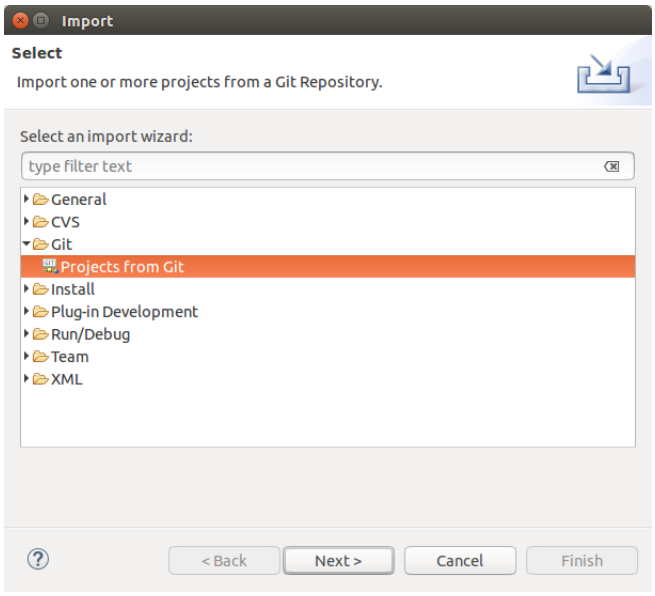


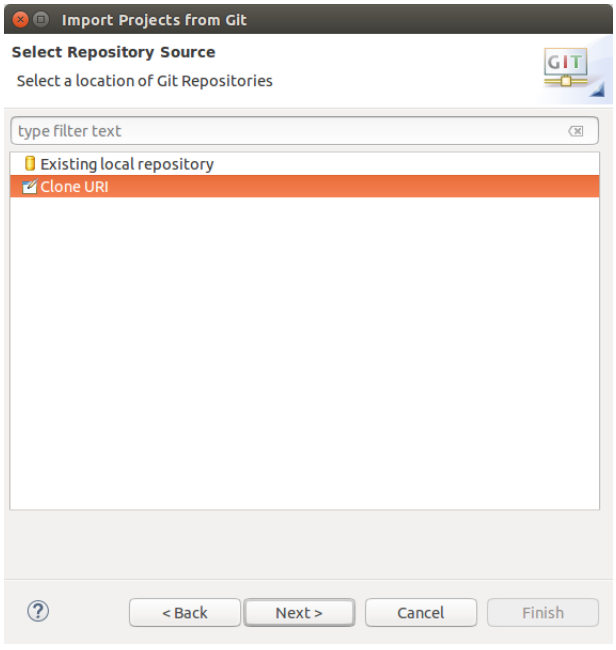
Select the correct project. This opens a new webpage. This page contains links for cloning the Git repository. The selection for HTTPS is depicted in the following screenshot. Please note that the URL for you is different, as it contains your user name.



13.2. Clone the repository via the Eclipse IDE

To clone, select *File → Import... → Git → Projects from Git → Clone URI*.

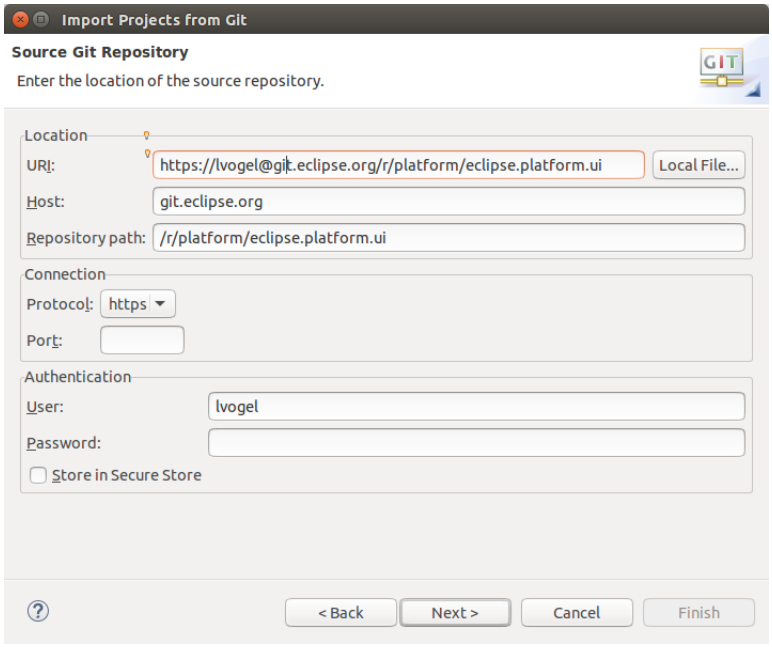




Enter the URL from the Gerrit webpage in the first line of the wizard. The Eclipse Git tooling removes "git clone" from the clone URI automatically.

Note that the URL automatically includes your user name. This makes it easier to push a change to the Eclipse Gerrit review system. If you clone the repository using the Eclipse Git functionality, your local repository is already configured to push changes to Gerrit, no additional setup step is required.

You can also clone anonymously via *Anonymous HTTP*, if you just want to play with the source code locally and do not intend to contribute changes. The following screenshot shows this selection, please the depicted user name with yours.



On the next wizard page you can choose to clone all branches or only the master branch. Finish this wizard by selecting the *Next* button until you the option *Finish* option becomes available. You end up with the projects from the repository imported into your workspace.

Note

In case you cloned a Git repository without using the Gerrit URL or not via Eclipse Git you have to adjust the push URL. See Section B.5, “Appendix: Cloning from the Git server and adjusting the push URL” for a description how to do that.

14
Install required plug-ins and configure your workspace

14.1. Exercise: Install additional plug-ins via install file

The Eclipse Platform UI team provides a file describing all the plugins typically needed for platform development. You can download from the following link this file, stored under version control: http://git.eclipse.org/c/platform/eclipse.platform.ui.git/plain/releng/org.eclipse.ui.releng/platformUiTools.p2f

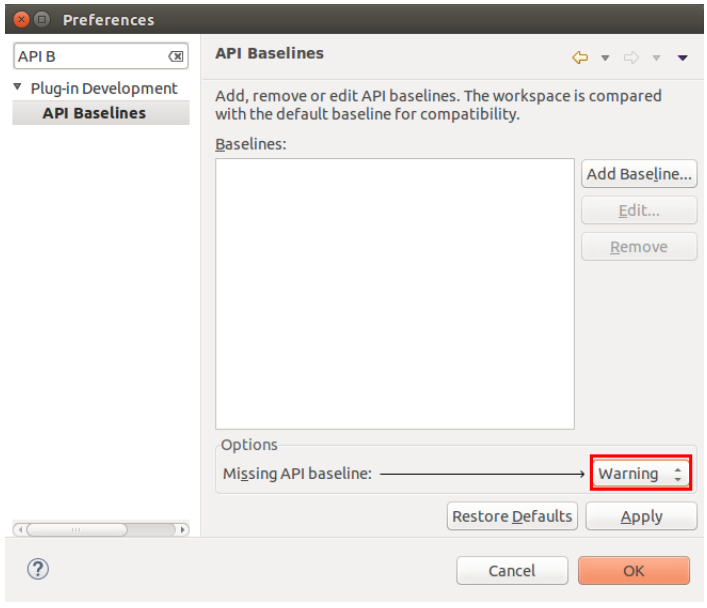
To install all the development plug-ins import the p2f file via the *File* → *Import...* → *Install* → *Install Software Items from File* menu path.

14.2. Configure API baseline

Eclipse projects frequently needs to ensure that they do not break API, compared to the previous release. For this the API tooling is used, and a missing API baseline is reported as error.

Tip

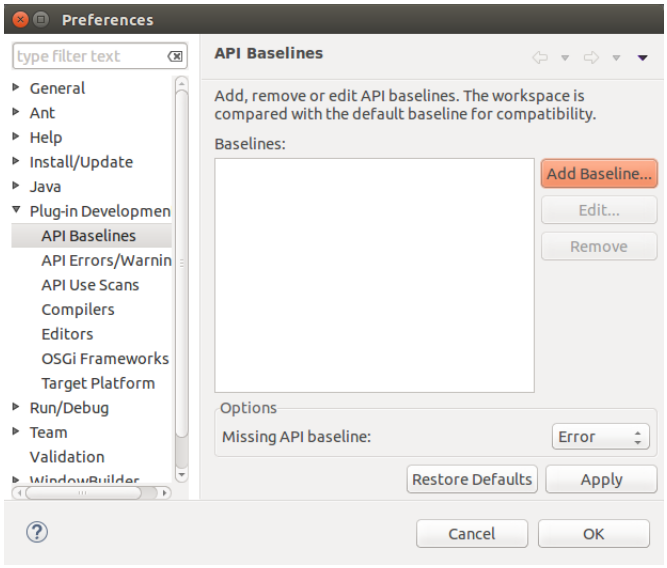
Handling baseline issues and API is something which the core committers can help with. New contributors can set this error message to warning to have an easy start. This setting is highlighted in the following screenshot.



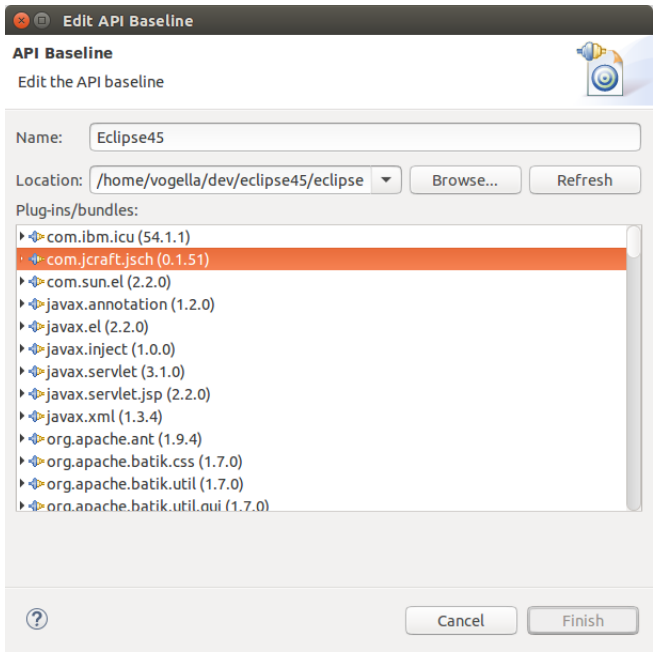
To define and use an API baseline, you need to download and install an official released version of Eclipse. This installation defines your baseline for the API, and you can specify its location in the Eclipse preferences. For this select the *Window* → *Preferences* → *Plug-in Development* → *API Baselines* menu entry. Here you can define the Eclipse release to use as baseline. With this system, Eclipse can report API breakage by comparing the code in your workspace with the last released code.

For example if you develop for the next Eclipse 4.x+1 release (for example 4.6), you should select the official Eclipse release (for example Eclipse 4.5.1) as API baseline. This way the Eclipse API baseline tooling can check if you break existing API and can add error or warning markers to your code. The tooling can also propose quickfixes for such violations.

The following screenshots demonstrates how to configure a API baseline. This assumes that the 4.5 is the latest official release. In the preference press the *Add Baseline...* button.



Enter a name for this baseline, select the folder which contains the latest official Eclipse release and press the *Reset* button to use this installation as baseline.



For detailed information about the usage of the API baseline, see *Eclipse Version Numbering Wiki [https://wiki.eclipse.org/Version_Numbering]* and *API Tools [https://wiki.eclipse.org/PDE/API_Tools/User_Guide]*.

14.3. Closing projects

Sometimes a plug-in give you error messages, which you can't solve. For example, if the plug-in is specific to the Windows operating system API and you are using Linux, you get compiler errors for it.

In this case, right-click on the project and select *Close Project* from the context menu. This will make the Eclipse IDE ignore this plug-in in your workspace.

14.4. Setup for other Eclipse projects

If you planning to contribute to another Eclipse project, please have a look at their contribution guide. This guide is typically available via the project website, which can be found via the search functionality of the *Eclipse project side [https://projects.eclipse.org]*.

15
Make a change and push the suggested change to Gerrit

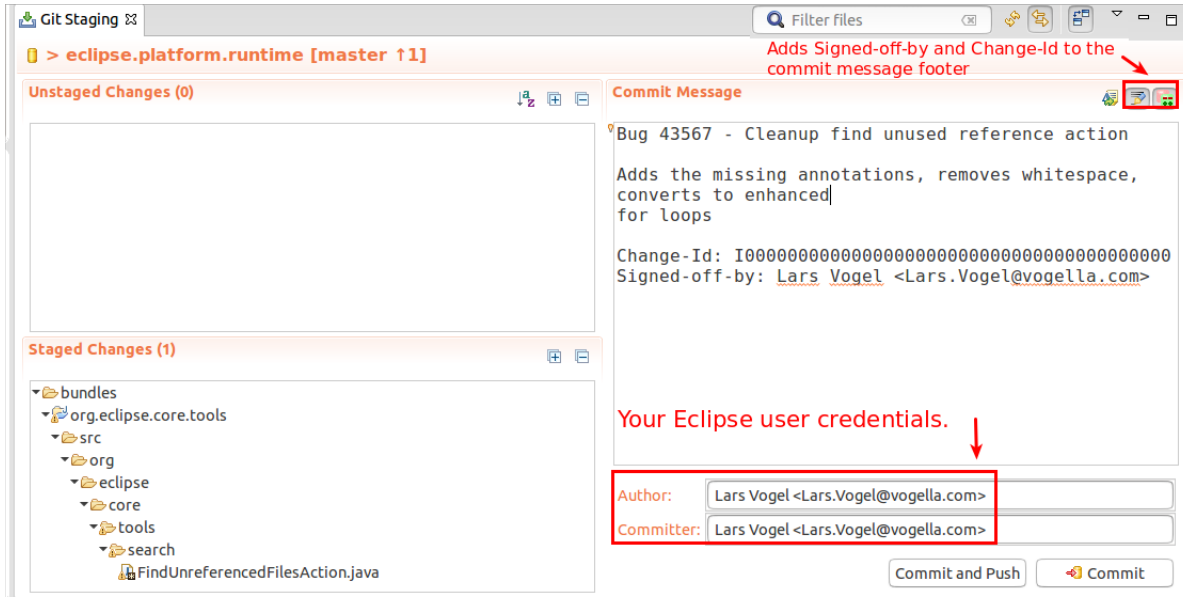
15.1. Make a code change

Find a very simple change for your first contribution. For example fix a typo or a compiler warning. Eclipse projects frequently tag bugs with the keyword *bugday* or *helpwanted*, these are candidates for initial contributions.

15.2. Push to the Gerrit review system

Use the *Git Staging* view to commit your change and to push it to the Gerrit review server.

You need to sign off every commit. The Git functionality in Eclipse simplifies that via the sign-off push button. The *Add Change-Id* push button allows you to include a Change-Id entry to the commit message. Both buttons are highlighted in the following screenshot.



Note

The Change-ID entry is initially set to "Change-Id: I0000000000000000000000000000000000000000". During the commit, this is replaced with an ID generated by the Git tooling. Like other Git IDs it is a SHA1 over the content it captures and the letter 'I' is added a prefix to make it obvious that it is a Gerrit Change-Id and not a Git commit ID.

Warning

To get picked up by Gerrit, the *Change-Id* must be in the last paragraph together with the other footers. To avoid problems do not separate it from the *Signed-off-by* message with a new-line character.

15.3. Updating the Gerrit review

You may get review comments asking to improve the Gerrit change. Or you find a better way of doing the change. You can update an existing Gerrit review by using the same Change-Id for another commit. The easiest way of doing this is to amend the last commit and push this commit again to Gerrit.

Since the amended commit contains the same Change-Id as the initial commit, Gerrit knows that you want to update this change with the new commit. Also in your local Git repository commit amend replaces the previous commit in your local history (and keep the same parent commit).

15.4. Learning Gerrit

The details of the Gerrit review system are coverted in Part III, “Using the Gerrit code review system”

Part III. Using the Gerrit code review system

16
Code review

16.1. The code review process

During a code review process a proposed change is reviewed by other developers. Every contributor can suggest changes and update the suggested changes. Once the change is accepted by a committer, it is merged into the code base.

For an efficient process it is important that the code review is conducted in a supportive environment where constructive feedback is given to enhance the change.

While a code review process can be implemented without any tool support, it is typically more efficient if a structured code review system is used. Gerrit is a code review system developed for the Git version control system.

It is also common practice to do an automated test build of the proposed merge using continuous integration tools like Jenkins or Hudson to check for compile time errors and to run a test suite.

16.2. Advantages of code review

In general a structured code review process has the following advantages:

1. Early error detection:
   - Build problems are detected immediately by automatic test builds.
   - Logical flaws can be spotted by the human reviewer before any code is merged.
2. Conformation to the source code standards:
   - Allows the team to identify early in the process any violations with the team code standards.
   - Helps to keep code readable and easier to maintain.
   - In some projects the continuous integration system also checks for code standard conformity automatically as part of the build process.
3. Knowledge exchange:
   - The code review process allows newcomers to see the code of other more experienced developers.
   - Developers get fast feedback on their suggested changes.
   - Experienced developers can help to evaluate the impact on the whole code.
4. Shared code ownership: by reviewing code of other developers the whole team gets a solid knowledge of the complete code base.

5. Easy entry to contribution: People without write permission to a repository have an easy way to start contributing and to get feedback.

6. Enables review before a change is submitted: Iteration based on review comments / build results happen before a change is merged into the target branch. This result in a cleaner history of the repository.

17
The Gerrit code review system

17.1. What is Gerrit?

Gerrit is a web based code review system, facilitating online code reviews for projects using the Git version control system. Gerrit is a Git server which adds a fine grained access control system and a code review system and workflow.

The code review system is web based, facilitating online code reviews for projects using the Git version control system. The user interface of Gerrit is based on *Google Web Toolkit* and its Git implementation is JGit.

A developer can use Gerrit to suggest a change. Other developers can review the change and suggest improvements. If a Gerrit change needs improvement, it is possible to update it with a new commit. Once the suggested changes are accepted by the reviewers, they can be merged to the target branch in the Git repository via Gerrit.

Gerrit makes code reviews easier by showing changes in a side-by-side display. It also supports to display the change as a unified diff which is often easier to read on smaller screens.

A reviewer can add comments to every single line changed.

The main entry point for Gerrit is the *Gerrit code review homepage [https://www.gerritcodereview.com/]*.

Gerrit is licensed under the Apache 2.0 license.

17.2. Change

The Gerrit review system uses the term *change*, to define a code review / review request. Each change is based on one commit.

17.3. How does Gerrit work?

Gerrit can prevent users from pushing directly to the Git repository. If you push to Gerrit, you use a certain path (*ref specification*) which tells Gerrit that you want to create a change. This push ref specification is *refs/for/master* if the target of the change under review is the master branch. You can also push to *refs/for/xyz* to put a commit into the review queue for the xyz branch.

If you push to this ref specification, Gerrit creates a new change or makes an update of an existing one. Gerrit uses the *Change-Id* information in the commit message to identify if the push is a new commit or an update of an existing change.

A change consists of one or more patch sets which are used to improve the first proposal based on review comments. One patch set corresponds to one Git commit.

It is still possible to bypass code review by pushing directly to *refs/heads/master* if sufficient rights have been granted.

17.4. Review categories - Verified and Code-Review

Gerrit supports different categories (also known as labels) for review feedback. In its default configuration it supports the *Code-Review* category.

In typical installations also the *Verified* category is installed.

The "Verified" category typically means you were able to build and test the change introduced with the Gerrit change. Typically, this is done by an automated process such as a Jenkins / Hudson build server.

The "Code-Review" category is typically used to vote on the quality of the implementation, code style, code conformity and that the overall design of the code is designed to the standards desired by the project.

Committers and contributors can vote in these categories.

17.5. Voting in Gerrit

The rules for voting in Gerrit is:

1. Highest vote (+2 in Code-Review and +1 in Verified) enables submitting
2. Lowest vote (-2 in Code-Review) is a veto blocking that the change can be submitted and can't be overruled by other reviewers
3. You cannot submit the changes to the Git repository until there is the highest vote in all categories

Typically Gerrit instances uses the *Verified* and the *Code-Review* category.

If you did some manual testing and the code works as you desire then it is good to +1 in the "Verified" category, or -1 if it failed some of your use cases that must be resolved in order to merge. Otherwise, leave it as 0 if you did not test the code.

Non-committers of the project can typically vote with -1 and +1 in Code-Review to indicate an opinion in either way but ultimately it is up to the decision of those with +2 power to make the overall decision. You may want to -2 vote to block the submission of the code if it is not up to par with your project's standards or +2 vote to indicate that you approve that the contribution is merged into the Git repository.

17.6. Gerrit online documentation

You find a quick introduction about Gerrit at: *Gerrit Code Review - A Quick Introduction [https://gerrit-review.googlesource.com/Documentation/intro-quick.html]* and detailed information in the *Gerrit documentation [https://gerrit-review.googlesource.com/Documentation/index.html]*.

Eclipse specific information can be found on the *Gerrit at Eclipse [http://wiki.eclipse.org/Gerrit]* webpage.

17.7. Posting draft reviews

There is also a special *refs/drafts/master* refspec which you can use to push changes to for private review before publishing them to all developers.

This is useful if your work is in progress and not ready for public review. You can also CC specific people to review this private change. Once it is ready for public review there is a *Publish* button to convert the draft into a change review.

18
Gerrit workflow description

18.1. A typical Gerrit workflow

The following describes a typical workflow of a developer who uploads a Gerrit review and a reviewer.

1. The developer fetches or pulls in the latest changes from the Git repository
2. The developer create a new local branch based on the `origin/master` branch. This step is optional but it is considered a good practice to create an independent branch per change to avoid unnecessary dependencies between commits (Gerrit reviews).
3. The developer implements a change(new feature, bug fix, documentation change) and create a commit in his local repository with these modifications.
4. The developer implements (if necessary) more changes and amends the existing commit, until he is satisfied with the change.

Afterwards you perform the following steps.

1. Push the change to the Gerrit review system to the `refs/for/master` refspec to create a change for the master branch.
2. If you receive improvement suggestions, fetch the latest changes and rebase your patch onto origin/master

Repeat the last three steps until the patch is finally accepted and merged into the codebase (or until the change is completely rejected). Finally you can delete your local branch.

18.2. Gerrit vs. Git workflow

In the Gerrit scenario amending a commit is used to update the Gerrit change request. Using the amend operation on a published commit is usually discouraged in Git, in Gerrit it is the normal process.

18.3. Gerrit vs. GitHub workflow

The Gerrit review system is based on the assumption that each commit is reviewed separately, while at GitHub, a complete branch is reviewed and merging with the pull request.

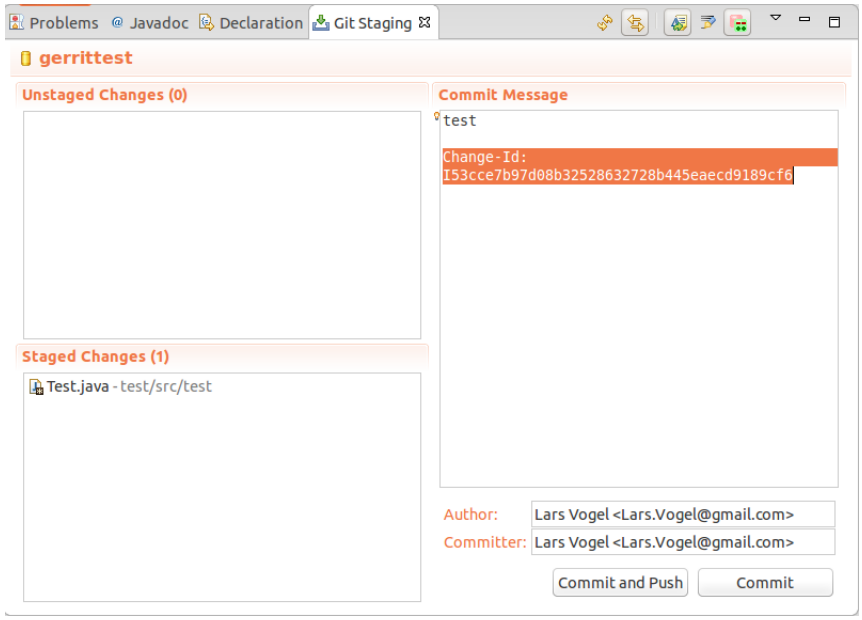
19
Working with Gerrit reviews

19.1. Creating and updating a Gerrit change request from Eclipse

In the following description we make a change in our local repository and create a Gerrit change request.

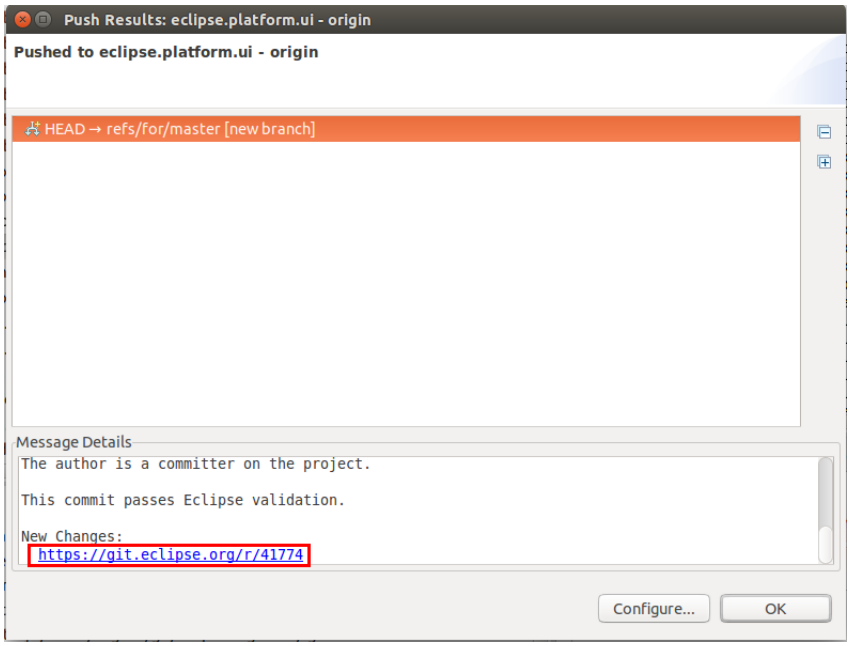
After getting the latest source code, when you are ready for development, create a new local branch starting from the origin/master branch.

Perform some changes in your Java project. Commit the changes and ensure to select the *Add Change-Id* button in the *Git Staging* view.

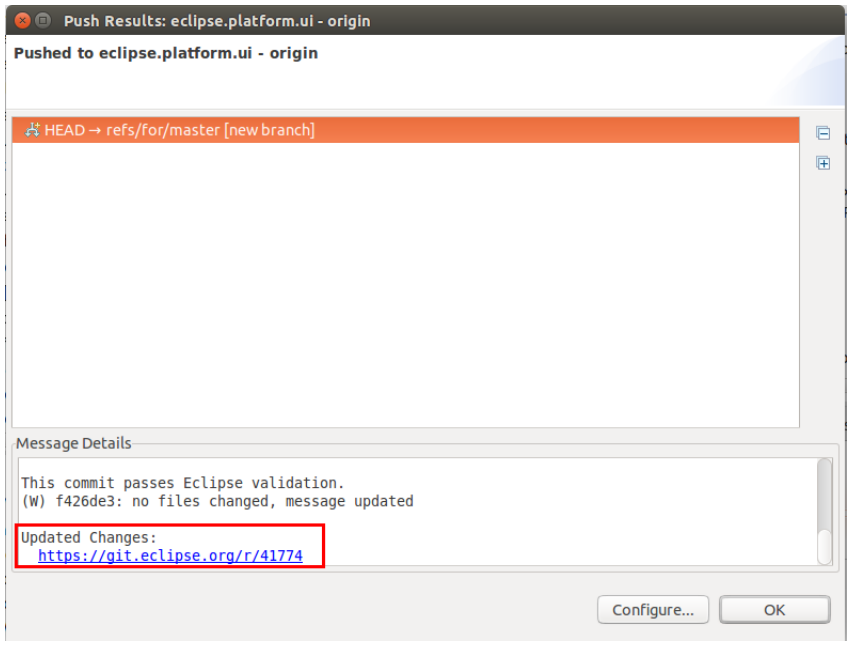


The Change-ID is what uniquely identifies the change in Gerrit. The entry is initially set to `Change-Id: I0000000000000000000000000000000000000000`. During the commit, this is replaced with an ID generated by the Git tooling.

Push the change to Gerrit. As your push configuration is configured for Gerrit, this should create a change. The Eclipse Git tooling shows a dialog after the push. This dialog contains error messages in case the push was not successful. Via the URL you can access the Gerrit web interface and analyze the Gerrit review.



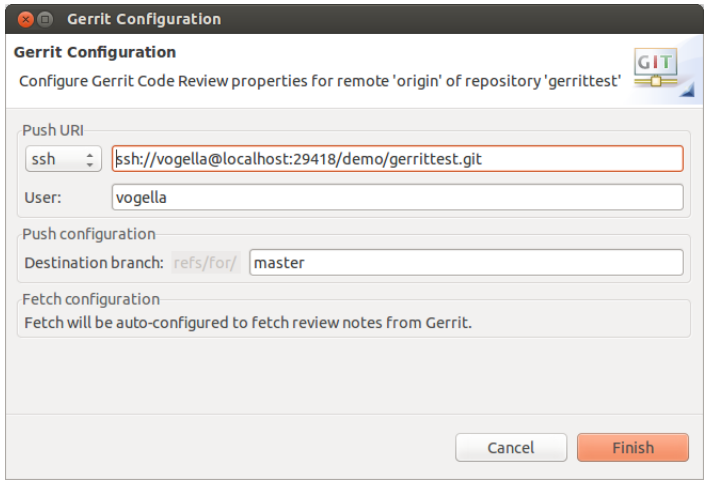
Do a few new changes to your Java project and commit them to your local Git repository. Ensure that you amend the existing commit. This makes sure the existing change id is used and avoids that Gerrit cannot apply your change because of a dependency cycle on the commits. Push your new commit to Gerrit, this should update the existing review request. The Eclipse Git push confirmation dialog should tell you that you created a new change set.



19.2. Gerrit support in Eclipse

To use Gerrit in Eclipse, clone the repository as seen before, and import your projects. By default, the Eclipse Git tooling creates a configuration that pushes the local master branch to the remote master branch. To connect with Gerrit, change the configuration to push the local master to `refs/for/master` instead.

After cloning the repository you can configure Gerrit in the *Git repositories* view by right-clicking on the *origin* remote and selecting *Gerrit Configuration*.



You can see the Gerrit review notes in Eclipse. Select *Open in Commit Viewer* on a commit in the History View. In the Commit Viewer you have an additional tab called *Notes*. Here you see the data which was created by the Gerrit reviewer.

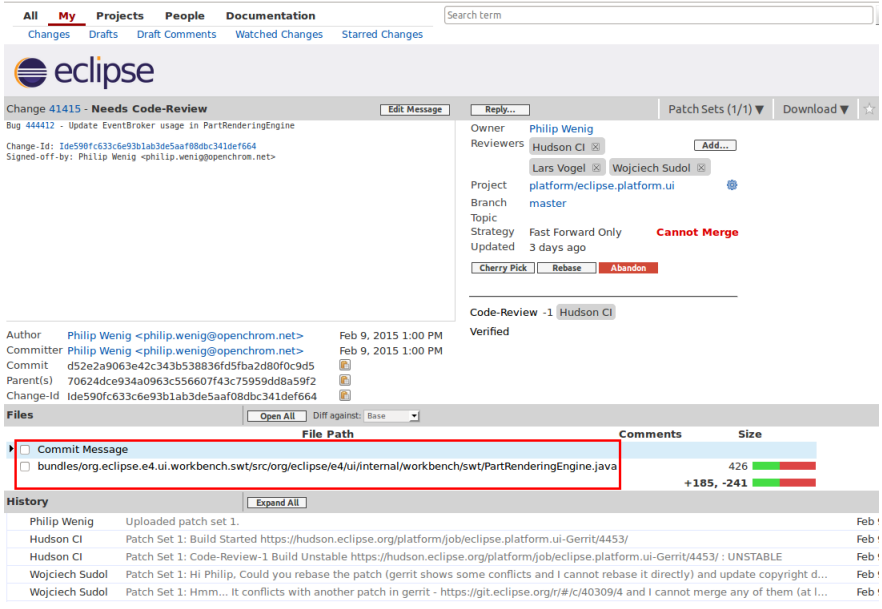
19.3. URL for reviews

To see the reviews on your Gerrit server, open the link to your Gerrit instance. For example if you have Gerrit locally installed use http://localhost:8080/ which is the default port for a local Gerrit installation. Or if you want to see the Gerrit reviews for the Eclipse project use *Eclipse Gerrit instance [https://git.eclipse.org/r/]*. You see all change requests, if you login into the Gerrit instance you see Gerrit changes you are involved with.

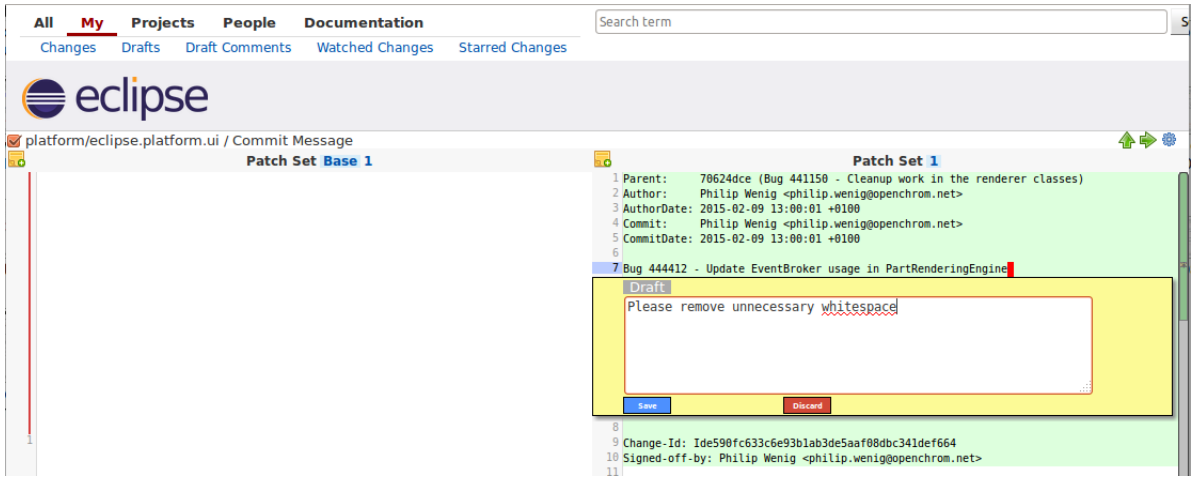
You can use the Search field to search for review see *Search options in Gerrit [https://git.eclipse.org/r/Documentation/user-search.html]* for available search options.

19.4. Reviewing a change request

Click on one change request to see the details of the Gerrit change request. Click on the commit message of the file name (highlighted in the screenshot) to see the changes in the file.



You can double-click on a line to comment on the change.

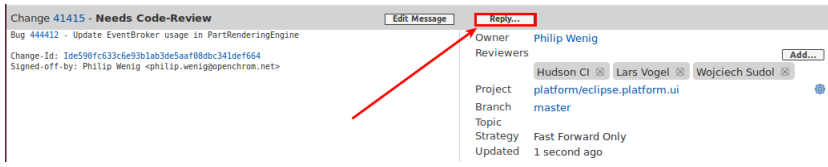


Clicking on *Up to change* brings you back to the change.

Gerrit allows you to review the commit message, which appears at the top of the file list. To understand the history of the repository it is important to provide a good commit message.

19.5. Publishing your review result

Click on *Reply* button to give feedback on the change.



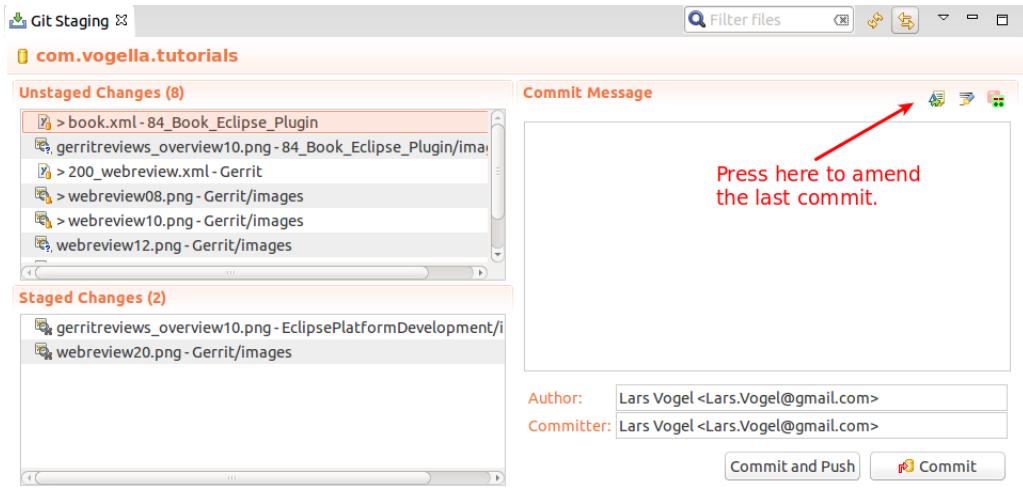
You can give a summary of your feedback and return a review number between -2 and +2 (if you are a committer) or between -1 and +1 (if you are not a committer). If you vote -1 or -2, you respectively indicate that the patch still requires rework or that you disagree with the suggested change. See Section 17.5, “Voting in Gerrit” for details of the default voting rules.

19.6. Improve a change based on review feedback

If the review process has resulted review comments explaining how to improve the proposed patch, the author of the patch (or someone else) can adjust the patch. The developer amends the commit in his local repository and pushes the improved commit to Gerrit. He can also adjust the commit message while amending the commit.

As the developer amends the commit, the same *Change-Id* is used and the Gerrit review system identifies the change as update of the existing change.

The *Git Staging* view allows amending the last commit by clicking the amend button shown in the screenshot.

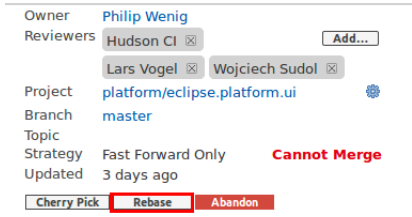


Tip

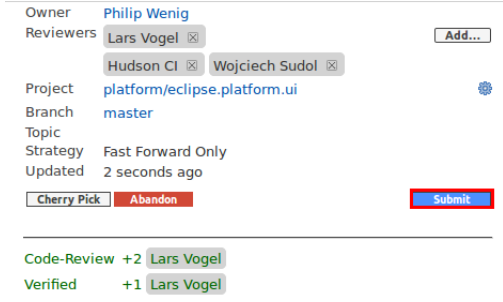
Gerrit allows you to select which patch sets you want to compare in *Comparison* view. Typically you compare the base version with the latest patch set, but if e.g. you already reviewed patch set 3 you may want to just see what the author changed between patchset 3 and patchset 5.

Once the change has reached the required quality, the reviewer can give a +2 evaluation of the change.

Depending on the settings of the Gerrit repository, the Gerrit review might have to be rebased. If you conflict happens during this rebase operation you can trigger this rebase directly from the web interface. If a change is rebased, it needs to be reviewed again.

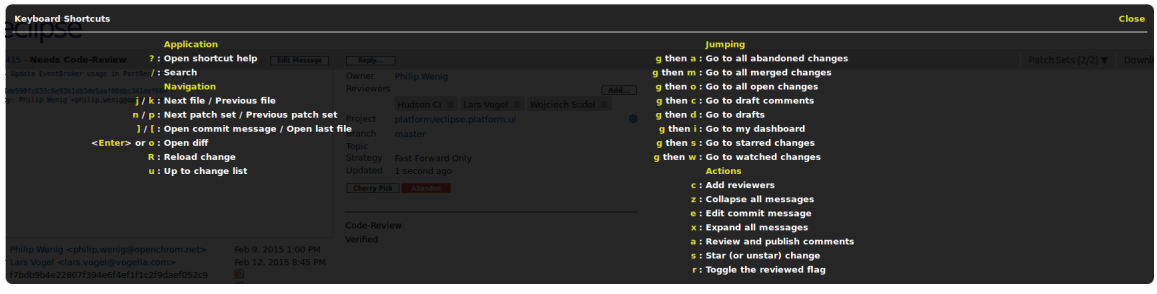


Afterwards a developer with commit rights can submit the change. This is typically done by the same committer who voted +2. Depending on the Gerrit configuration for the project, this may only work if the change is based on the current origin/master branch. Gerrit allows the committer and contributor to rebase the change if that is not the case.



19.7. Keyboard shortcuts in Gerrit

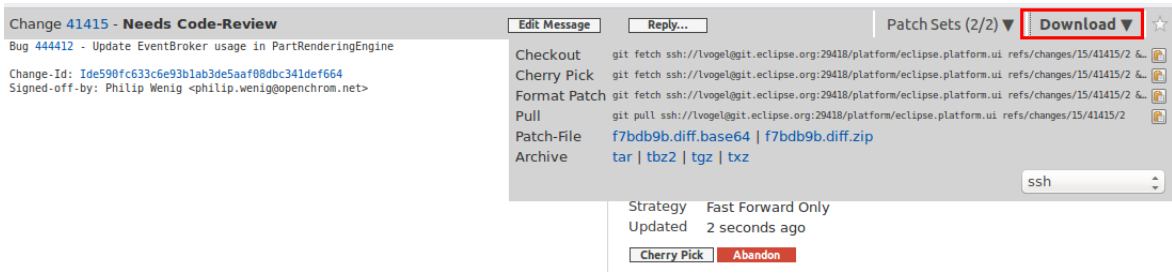
Press the "?" key in the Gerrit web user interface to see the actions you can trigger via shortcuts. The available shortcuts depend on there you are in the user interface.



The available shortcuts are different, depending on the page your are.

19.8. Fetching a remote change into a local Git repository

It is possible to fetch the changes from the Gerrit review into another local Git repository. The Gerrit page lists the commands for downloading the change. This is depicted in the following screenshot.

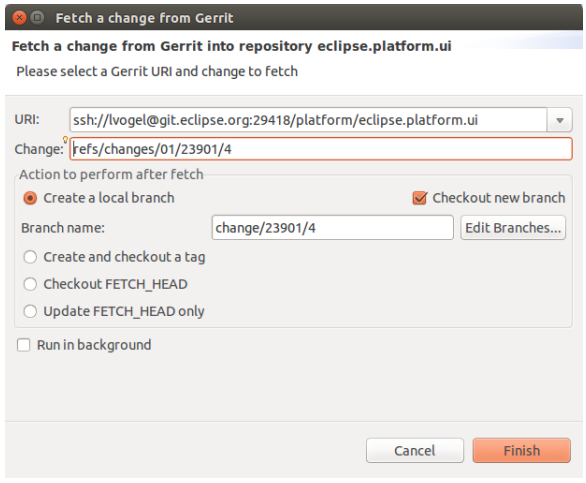


After fetching a change, the developer can adjust the change and amend the commit. If he pushes it to Gerrit, the change is updated. Such a procedure should be coordinated by the author of the original change to avoid that two developers do the same work.

19.9. Fetching a remote change with Eclipse Git

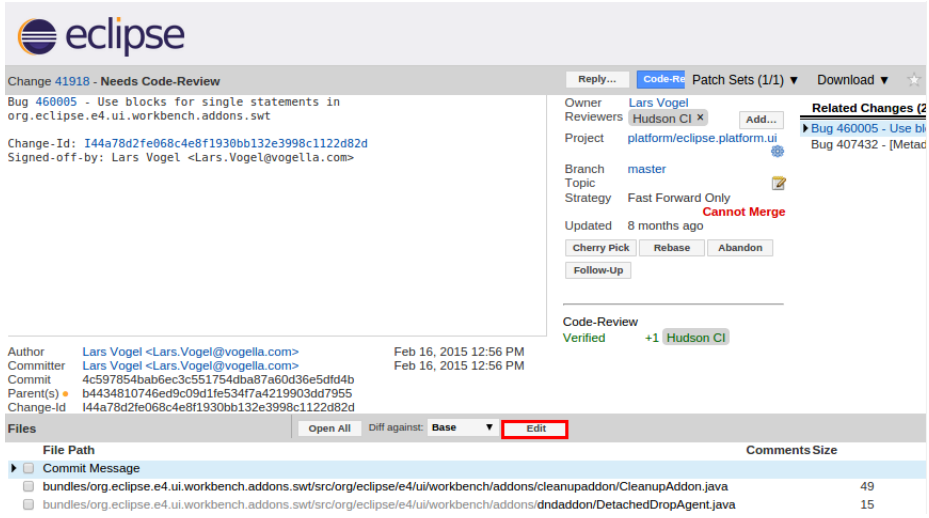
Tip

The Git tooling in Eclipse allows you to fetch a change from Gerrit. For this, right-click on the project *Team → Remote → Fetch from Gerrit*. When working with many projects, it is often easier to right-click on the repository instead and access *Fetch from Gerrit* from there. You can paste the change's number (shown in the change's URL) and use **Ctrl**+**Space** to expand the change number to the list of patch sets, select the one you want to download and fetch it into a new local branch. Afterwards you can to test the changes in isolation.



19.10. Edit a change directly via the web interface

You can also edit directly in Gerrit. Press the edit button for this.



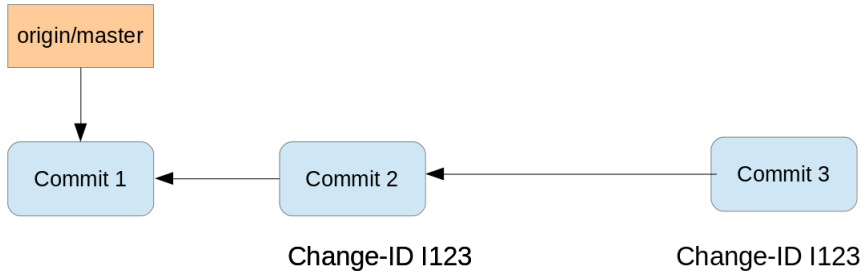
Afterwards select the file or the commit message you want to edit. Press save and close once you are done. After you are done with all your edits, press the *Publish Edit*.

20 Solving technical Gerrit problems and general watchouts for contributions

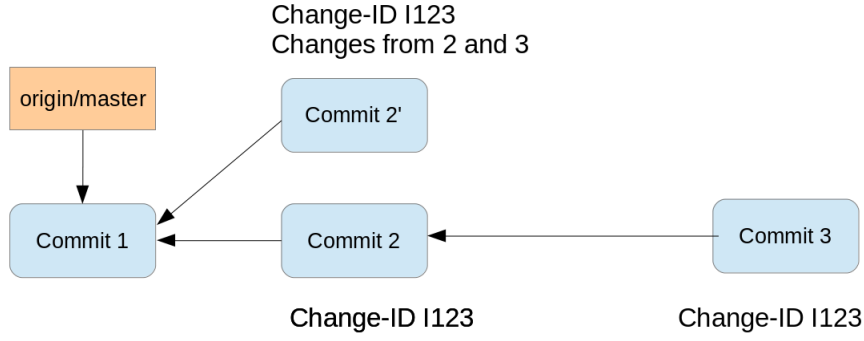
20.1. You forgot to amend the commit

If you try to push a commit which contains the same ChangeId as a predecessor commit, Gerrit will reject it and responds with the error message: "squash commits first".

In this case you probably forgot to amend the existing commit but used the same Gerrit Change-ID. This is depicted in the following graphic.



In this case you need to squash the commits. This results in a new commit which can be pushed to Gerrit. As you use the same Change-ID this pushed commit will update the Gerrit review. In the following diagram the "Commit 2'" is the result of squashing "Commit 2" and "Commit 3" into one commit.



An easy solution to handle this is, to do a soft reset in Git to the previous commit and commit the change files again, this time with the amend option. If you really intended to create two commits you need to generate a new changeId in the second commit which was rejected.

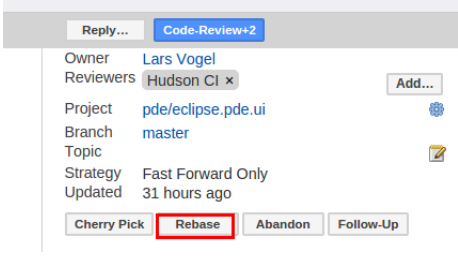
Tip

The Git tooling in Eclipse allows squashing commits via the *Git Interactive Rebase* view. A simple way of doing this is by selecting the adjacent commits you want to squash in the *History* view and by selecting *Modify* → *Squash* from the context menu.

20.2. Handling merge conflicts

The submit step may fail due to merge conflicts, depending on the Gerrit project configuration.

The easiest way to do so is via the *Rebase* button in the Gerrit web interface, if there are no conflicts.



In case you have to resolve merge conflicts you have to do this locally and push an updated commit. The steps required to solve a merge conflict are:

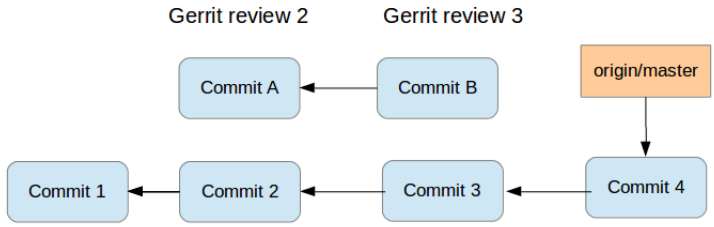
- Rebase your local branch onto the latest state of `origin/master`
- Resolve all conflicts
- Commit them using *Rebase* → *Continue*.
- Push your change again to Gerrit for review

This creates a new patch set for the change. The new patch set has to pass the code review again. Submit the new patch set change to merge it into the master branch.

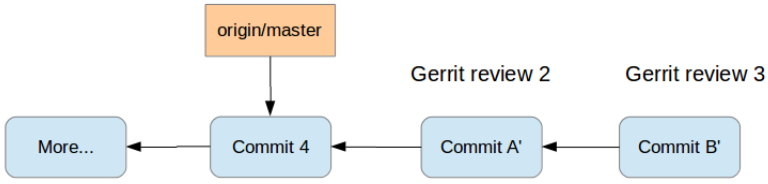
20.3. non-fast forward

You get this error message if you try to submit a commit in Gerrit to the Git repository and the change would result in a non-fast forward merge. The Gerrit service is sometimes configured with the setting to allow only fast-forward merges. The default submit type in Gerrit is "Merge if necessary".

This is the case if the pushed commit is not based on the current tip of the remote branch, e.g., origin/master if you aim to merge your change into this branch.. This problem is depicted in the following graphic.



The solution is to rebase your commit onto `origin/master`. This updates your change onto changes which reached origin/master while you were working on your change. Afterwards you push the commit again to Gerrit.

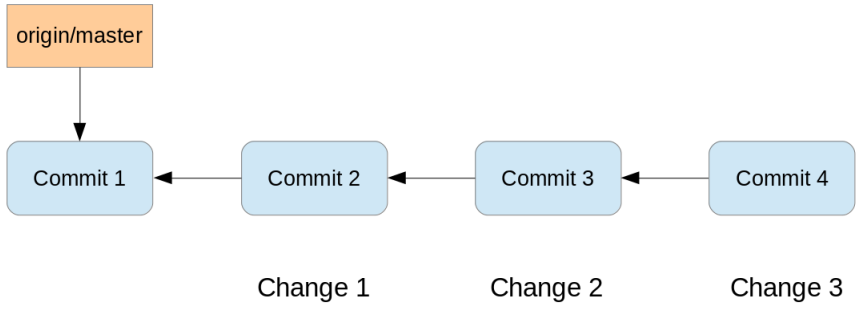


See *non-fast forward [https://git.eclipse.org/r/Documentation/error-non-fast-forward.html]* for details.

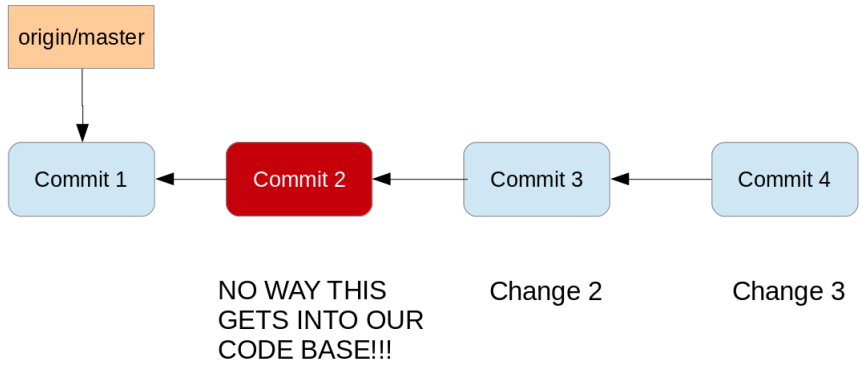
20.4. Remove a bad commit from a series of commits

If you create a series of dependent commits, it might happen that one of the commits is rejected during the review process.

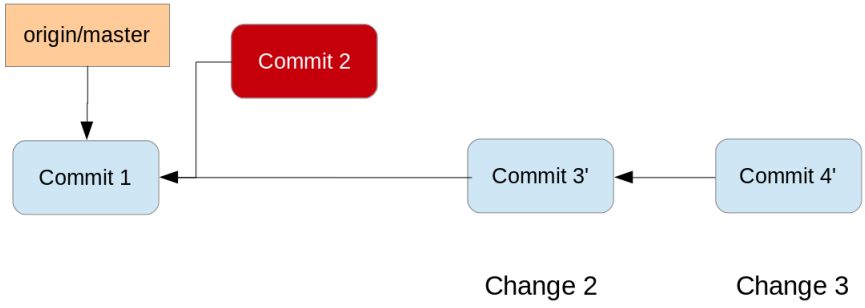
In this case you cannot merge the other commits as they still have a dependency to the "bad" commit.



During the review process commit 2 are rejected. The commit 3 and commit 4 are still good and do not depend on the changes in commit 2.



The solution is to use interactive rebase to skip the bad commit.



Tip

The Gerrit user interface does show "depends on" and "needed by" fields when it knows that a Gerrit review depends on another review. This helps you manage your dependency trail.

20.5. More error messages and their solution

See *Gerrit error message [https://git.eclipse.org/r/Documentation/error-messages.html]* for more error messages and their solutions.

20.6. Things to consider if you want to contribute to an existing project

20.6.1. How to start contributing?

It is typically a good approach to fix something really simple as the first step. Search for a few warning messages in the code, or a typo in the documentation, and prepare a fix for it.

Once you get familiar with the contribution process start contributing something more difficult.

20.6.2. Scope of a Gerrit change / commit

A Gerrit change (== one commit) should address only one issue.

Avoid going "off topic" with your patches. If you would like to clean up some code, or fix something out of the scope of your patch, create additional patches. This will make your patches more concise, easier to review and easier to revert if something go wrong.

In many open source projects smaller patches tend to be reviewed faster and merged faster as it is easier for the committer to review. With this in mind, try to break your contributions into small concise Gerrit changes. It is more likely that these get reviewed and accepted.

It also helps to reach a clean history. Imagine you are chasing a bug in code which was introduced 2 years back. If the history is built from small concise commits, it is easier to find the commit which introduced the problem and understand the motivation why it was done in the way it was implemented.

20.6.3. Commit message

Most project handles the required commit message a bit differently. For example, the Eclipse Platform project prefers the following format:

```
Bug XXXXXXXX - bug title

Short description of what the fix contains, or the direction of the fix

Change-Id: I0000000000000000000000000000000000000000
Signed-off-by: email-from-the-CLA
```

The text in the header must match the title in Bugzilla. The Eclipse Git tooling complains about commit headers longer than 80, but you can ignore this warning. For example the following is a valid commit message for the Eclipse platform project. The description in the body of the commit message can take as many lines as necessary and should be descriptive of what is going into the patch. For more complex patches you should provide implementation details and enough notes to allow the reviewer to understand what your fix provides.

```
Bug 434846 - Update the system default Locale if the -nl Locale is invalid

An incorrect locale value causes the IEclipseContext to run the conversion frequently.
The patch ensures that only valid values are saved in Eclipse context.

Change-Id: I6099172a5986e9678a830f84d78b2cdb8c161dc6
Signed-off-by: Dirk Fauth <dirk.fauth@fake.com>
```

It is important to explain in the commit body, why the change was done. Also explain why you have chosen a certain implementation method, if there was something special about it.

Tip

Don't worry too much about the correct commit message format, the project you are contributing to will also review your commit message.

20.6.4. Copyright header update

In some projects the source file contains a copyright header. If you modify the file you should also update the copyright header. Different project might use different standards here. Have a look at existing commits and the header in the existing files and try to adapt the style preferred by the existing developers.

20.6.5. Example: Copyright header in Eclipse Platform UI

The Eclipse platform project for example requires you to update the copyright header with the last modified year. Optionally you can also add yourself to the copyright with a reference to the bug. The following listing shows an example for this standard.

```
/*******************************************************************************
* Copyright (c) 2008, 2015 IBM Corporation and others.
* All rights reserved. This program and the accompanying materials
* are made available under the terms of the Eclipse Public License v1.0
* which accompanies this distribution, and is available at
* http://www.eclipse.org/legal/epl-v10.html
*
* Contributors:
*     IBM Corporation - initial API and implementation
*     John Doe <John.Doe@fake.org> - Bug 429728 Fix compiler warning
*******************************************************************************/
```

The above file was created in 2008 and last modified in 2015. It was originally created by employees of IBM and also modified by John Doe and he fixed Bug 429728 with it. The additional entry in the *Contributors* section is typically considered optional, but this depends on the individual project.

If you work for an organization the Foundation recommends adding the organization name in parentheses. See: *Copyright info from eclipse.org [http://www.eclipse.org/legal/copyrightandlicensenotice.php]*. For example:

```
John Doe (vogella GmbH) <John.Doe@fake.com> - Bug 429728 - Fixed compiler warnings
```

If you add a new file don't forget to add the copyright header using the project's license. For new files, list "yourself and others" instead of "IBM and others" in the first line.

If you fix several bugs in the same file, you should list the bug separately as in the following example listing. In case you find yourself changing the same file over and over again, you could also use "ongoing maintenance" instead of listing every bug number.

```
/*******************************************************************************
* Copyright (c) 2008, 2014 IBM Corporation and others.
* All rights reserved. This program and the accompanying materials
* are made available under the terms of the Eclipse Public License v1.0
* which accompanies this distribution, and is available at
* http://www.eclipse.org/legal/epl-v10.html
*
* Contributors:
*     IBM Corporation - initial API and implementation
```

```
*      John Doe <John.Doe@eclipse.org> - Bug 429728, 429972
*******************************************************************************/
```

In most cases you should avoid putting in the bug number into the source code (except the header). Only if you have to implement a workaround for an existing bug, you should add a reference in the source to this bug.

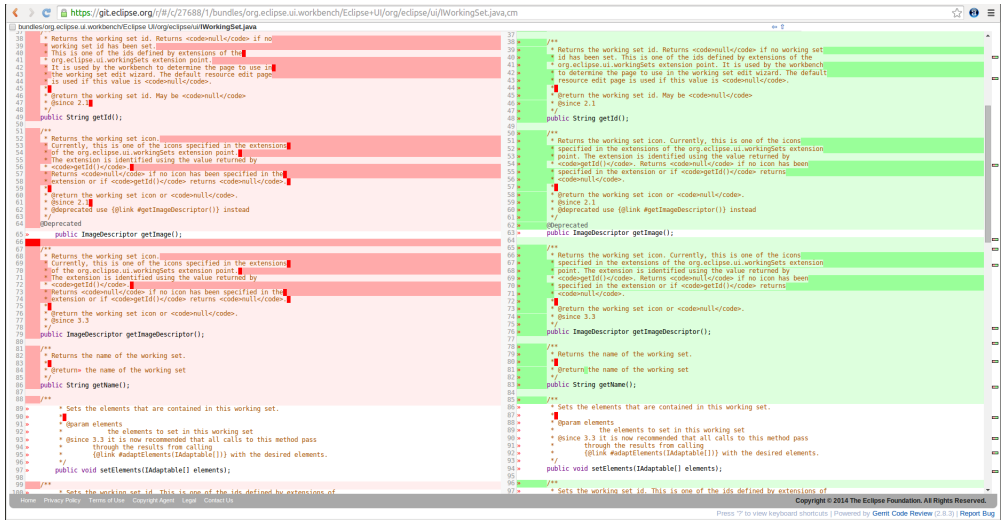
20.6.6. Stay polite

The best way to get your contribution accepted is to stay polite and to follow the guidance of the existing committers. For the first few contributions it is best to stay with the current processes and guidelines. If you continue to contribute, you will gain more trust and responsibility and you may be able to improve also the processes used by the project.

20.6.7. Avoid unnecessary formatting changes

Typically open source projects dislike if you re-format unchanged lines, as this makes the review process much harder. Avoid pressing the *Format source code* shortcut in your editor. Instead, only select the changes lines and then press the format source code shortcut (this is also an auto-save preference).

Unnecessary format changes makes it really hard to review a Gerrit change. The following screenshot shows a file which was reformatted, it is unnecessary hard for the reviewer to find the real change.



20.6.8. Avoid unnecessary whitespace changes

Typically projects dislike unnecessary whitespace in the contributed code. Ensure that you remove trailing whitespaces from your contributed code.

Unfortunately lots of project have unnecessary whitespace included in the source code, most committers also don't like if you remove them as it changes the history.

20.6.9. Pushing unfinished features

For most projects it is OK to push an initial (unfinished) implementation concept to Gerrit and to ask for feedback.

Mark unfinished commits with [RFC] (request for comments) or [WIP] (work in progress) in the commit message header to avoid frustrating reviewers who typically want to know if the change is considered to be ready by the author or not.

Tip

In general you should develop in the open as much as possible to avoid the unpleasant surprise that your perfect contribution is rejected because it does not fit into the design of the software.

20.6.10. Valid user and email

Some project uses additional validation plug-ins. For example the Eclipse foundation uses a server-side Gerrit plugin which checks for conformity with the CLA. When you push a change to Gerrit this plugin verifies if your contribution passes all the CLA requirements from the Eclipse foundation.

This means that the author field must be a valid Eclipse user. If the author is not a committer it requires that the author also "Sign-off" their contribution. Eclipse Git provides a preference to always sign-off changes.

See: *Wiki for Git contributions [https://wiki.eclipse.org/Development_Resources/Handling_Git_Contributions]*

20.6.11. Not getting feedback

If everything works perfectly, the developers of the project you are contributing to will accept your contribution and help you to solve all your open questions.

If you don't get feedback there are different possible reasons: maybe all committers are busy or in vacation. Or your change is extremely large (hard to review) or it ignores all the rules explained in the project's contributor guide.

If your contribution is ignored, feel free to ask in the Bug report for feedback or to send email to the project mailing list.

20.6.12. Dealing with negative feedback

Don't get frustrated by negative feedback, as the collaboration almost always results in a better commit that benefits everyone.

21
Reviewing changes

21.1. Doing a review

If you upload a new change to Gerrit it is good practice to validate the Gerrit change request yourself, to see if you find any obvious errors.

Tip

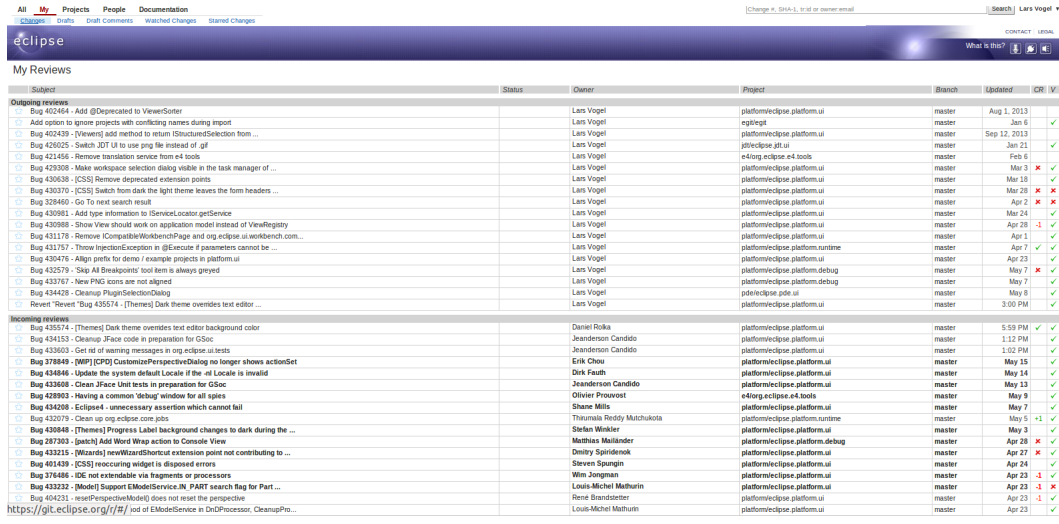
Reviews can be done by everybody with a valid Gerrit user account, i.e., you do not have to be a committer to perform Gerrit reviews. Doing code reviews of existing change requests is a good practice for new contributors to learn about the project.

To contribute to the project, reviewing is as important as writing new code or filling bug reports.

21.2. Seeing your reviews

You find all changes under the following URL: *https://git.eclipse.org/r/*.

If you log into Gerrit you see all your Gerrit changes and the reviews you gave feedback on or changes you were invited for review by another user.



Tip

You can also search for changes, for example by project. See *User search guide [https://git.eclipse.org/r/Documentation/user-search.html]* for details.

Part IV. Introduction to unit testing and the Eclipse test suites

22
Unit tests, integration tests and performance tests

22.1. What are software tests?

A software test is a piece of software, which executes another pierce of software and validates if that code results in the expected state (state testing) or executes the expected sequence of events (behavior testing).

22.2. Why are software tests helpful?

Software unit tests help the developer to verify that the logic of a piece of the program is correct.

Running tests automatically helps to identify software regressions introduced by changes in the source code. Having a high test coverage of your code allows you to continue developing features without having to perform lots of manual tests.

23
Testing terminology

23.1. Code (or application) under test

The code which is tested is typically called the *code under test*. If you are testing an application, this is called the *application under test*.

23.2. Test fixture

A *test fixture* is a fixed state in code which is tested used as input for a test. Another way to describe this is a test precondition.

For example, a test fixture might be a a fixed string, which is used as input for a method. The test would validate if the method behaves correctly with this input.

23.3. Unit tests and unit testing

A *unit test* is a piece of code written by a developer that executes a specific functionality in the code to be tested and asserts a certain behavior or state.

The percentage of code which is tested by unit tests is typically called *test coverage*.

A unit test targets a small unit of code, e.g., a method or a class, (local tests). External dependencies should be removed from unit tests, e.g., by replacing the dependency with a test implementation or a (mock) object created by a test framework.

Unit tests are not suitable for testing complex user interface or component inteaction. For this you should develop integration tests.

23.4. Integration tests

An *integration test* has the target to test the behavior of a component or the integration between a set of components. The term *functional test* is sometimes used as synonym for integration test. Integration tests check that the whole system works as intended, therefore they are reducing the need for intensive manual tests.

This kind of tests allow you to translate your user stories into a test suite, i.e., the test would resemble an expected user interaction with the application.

23.5. Performance tests

Performance tests are used to benchmark software components repeatedly. Their purpose is to ensure that the code under test runs fast enough even if it's under high load.

23.6. Behavior vs. state testing

A test is a behavior test (also called interaction test) if it does not validate the result of a method call, but checks if certain methods were called with the correct input parameters.

State testing is about validating the result, while behavior testing is about testing the behavior of the application under test.

If you are testing algorithms or system functionality, you want to test in most cases state and not interactions. A typical test setup uses mocks or stubs of related classes to abstract the interactions with these other classes away and tests state in the object which is tested.

24
Unit testing with JUnit

24.1. The JUnit framework

JUnit in version 4.x is a test framework which uses annotations to identify methods that specify a test.

The main websites for JUnit are the *JUnit homepage [http://junit.org/]* and the *GitHub project page [https://github.com/junit-team/junit]*.

24.2. How to define a test in JUnit?

A JUnit *test* is a method contained in a class which is only used for testing. This is called a *Test class*.

To write a test with the JUnit 4.x framework you annotate a method with the `@org.junit.Test` annotation.

In this method you use an *assert* method, typically provided by the JUnit or another assert framework, to check the expected result of the code execution versus the actual result. These method calls are typically called *asserts* or *assert statements*.

You should provide meaningful messages in assert statements so that it is easier for the developer to identify the problem. This helps in fixing the issue, especially if someone looks at the problem, who did not write the code under test or the test code.

24.3. Example JUnit test

The following code shows a JUnit test. This test assumes that the `MyClass` class exists and has a `multiply(int, init)` method.

```
import static org.junit.Assert.assertEquals;

import org.junit.Test;

public class MyTests {

  @Test
  public void multiplicationOfZeroIntegersShouldReturnZero() {

    // MyClass is tested
    MyClass tester = new MyClass();

    // assert statements
    assertEquals("10 x 0 must be 0", 0, tester.multiply(10, 0));
    assertEquals("0 x 10 must be 0", 0, tester.multiply(0, 10));
    assertEquals("0 x 0 must be 0", 0, tester.multiply(0, 0));
  }

}
```

24.4. JUnit naming conventions

There are several potential naming conventions for JUnit tests. In widespread use is to use the name of the class under test and to add the "Test" suffix to the test class.

For the test method names it is frequently recommended to use the word "should" in the test method name, as for example "ordersShouldBeCreated" or "menuShouldGetActive" as this gives a good hint what should happen if the test method is executed.

As a general rule, a test name should explain what the test does so that it can be avoided to read the actual implementation.

24.5. JUnit naming conventions for Maven

If you are using the Maven build system, you should prefer the "Test" suffix over "Tests" as the Maven build system (via its surfire plug-in) automatically includes such classes in its test scope.

24.6. JUnit test suites

If you have several test classes, you can combine them into a *test suite*. Running a test suite will execute all test classes in that suite in the specified order.

The following example code shows a test suite which defines that two test classes (MyClassTest and MySecondClassTest) should be executed. If you want to add another test class you can add it to `@Suite.SuiteClasses` statement.

```
package com.vogella.junit.first;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses({ MyClassTest.class, MySecondClassTest.class })
public class AllTests {

}
```

A test suite can also contain other test suites.

24.7. Run your test from the command line

You can also run your JUnit tests outside Eclipse via standard Java code. Build frameworks like Apache Maven or Gradle in combination with a Continuous Integration Server (like Hudson or Jenkins) are typically used to execute tests automatically on a regular basis.

The `org.junit.runner.JUnitCore` class provides the `runClasses()` method which allows you to run one or several tests classes. As a return parameter you receive an object of the type `org.junit.runner.Result`. This object can be used to retrieve information about the tests.

The following class demonstrates how to run the MyClassTest. This class will execute your test class and write potential failures to the console.

```
package de.vogella.junit.first;

import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class MyTestRunner {
  public static void main(String[] args) {
    Result result = JUnitCore.runClasses(MyClassTest.class);
    for (Failure failure : result.getFailures()) {
      System.out.println(failure.toString());
    }
  }
}
```

This class can be executed like any other Java program on the command line. You only need to add the JUnit library JAR file to the classpath.

25
Basic JUnit code constructs

25.1. Available JUnit annotations

JUnit 4.x uses annotations to mark methods as test methods and to configure them. The following table gives an overview of the most important annotations in JUnit.

Table 25.1. Annotations

Annotation	Description
@Test public void method()	The `@Test` annotation identifies a method as a test method.
@Test (expected = Exception.class)	Fails if the method does not throw the named exception.
@Test(timeout=100)	Fails if the method takes longer than 100 milliseconds.
@Before public void method()	This method is executed before each test. It is used to prepare the test environment (e.g., read input data, initialize the class).
@After public void method()	This method is executed after each test. It is used to cleanup the test environment (e.g., delete temporary data, restore defaults). It can also save memory by cleaning up expensive memory structures.
@BeforeClass public static void method()	This method is executed once, before the start of all tests. It is used to perform time intensive activities, for example, to connect to a database. Methods marked with this annotation need to be defined as `static` to work with JUnit.
@AfterClass public static void method()	This method is executed once, after all tests have been finished. It is used to perform clean-up activities, for example, to disconnect from a database. Methods annotated with this annotation need to be defined as `static` to work with JUnit.
@Ignore or @Ignore("Why disabled")	Ignores the test method. This is useful when the underlying code has been changed and the test case has not yet been adapted. Or if the execution time of this test is too long to be included. It is best practice to provide the optional description, why the test is disabled.

25.2. Assert statements

JUnit provides static methods in the `Assert` class to test for certain conditions. These *assert statements* typically start with `assert` and allow you to specify the error message, the expected and the actual result. An *assertion method* compares the actual value returned by a test to the expected value, and throws an `AssertionException` if the comparison test fails.

The following table gives an overview of these methods. Parameters in [] brackets are optional and of type String.

Table 25.2. Methods to assert test results

Statement	Description
fail(message)	Let the method fail. Might be used to check that a certain part of the code is not reached or to have a failing test before the test code is implemented. The message parameter is optional.
assertTrue([message,] boolean condition)	Checks that the boolean condition is true.
assertFalse([message,] boolean condition)	Checks that the boolean condition is false.
assertEquals([message,] expected, actual)	Tests that two values are the same. Note: for arrays the reference is checked not the content of the arrays.
assertEquals([message,] expected, actual, tolerance)	Test that float or double values match. The tolerance is the number of decimals which must be the same.
assertNull([message,] object)	Checks that the object is null.
assertNotNull([message,] object)	Checks that the object is not null.
assertSame([message,] expected, actual)	Checks that both variables refer to the same object.
assertNotSame([message,] expected, actual)	Checks that both variables refer to different objects.

25.3. Test execution order

JUnit assumes that all test methods can be executed in an arbitrary order. Well-written test code should not assume any order, i.e., tests should not depend on other tests.

As of JUnit 4.11 the defaulwhich may vary from run to run.t is to use a deterministic, but not predictable, order for the execution of the tests

You can use an annotation to define that the test methods are sorted by method name, in lexicographic order. To activate this feature, annotate your test class with the `@FixMethodOrder(MethodSorters.NAME_ASCENDING)` annotation. You can also explicitely set the default by using the `MethodSorters.DEFAULT` parameter in this annotation. You can also use `MethodSorters.JVM` which uses the JVM defaults, which may vary from run to run.

26
Eclipse and tests

26.1. Eclipse projects and their test suite

Most Eclipse projects provide a test suite which can be used to validate that everything still works as planned.

For example, in case you are changing the Eclipse platform code you should ensure that the platform tests still perform as planned. If changes in the tests are required, you should adjust the tests with the same Gerrit change.

26.2. Version of JUnit

At the time of this writing, lots of unit tests of the Eclipse projects are still based on JUnit 3.x. JUnit 3.x uses the *test* prefix of method names to identify test methods.

Several projects are currently migrating their test to JUnit 4, so this will hopefully change in the future.

26.3. Tips: running the unit tests on a virtual server

On Unix based system you can also run the user interface tests with a virtual display. This makes the execution of the tests faster and allows to developer to continue to work on the same machine.

> **Tip**
>
> This step is optional. You can of course run the tests without a virtual server.

The Eclipse platform unit tests starts an Eclipse IDE and visually interacts with it. This screen flickering can be annoying. You can use a virtual server and execute the tests on this virtual device.

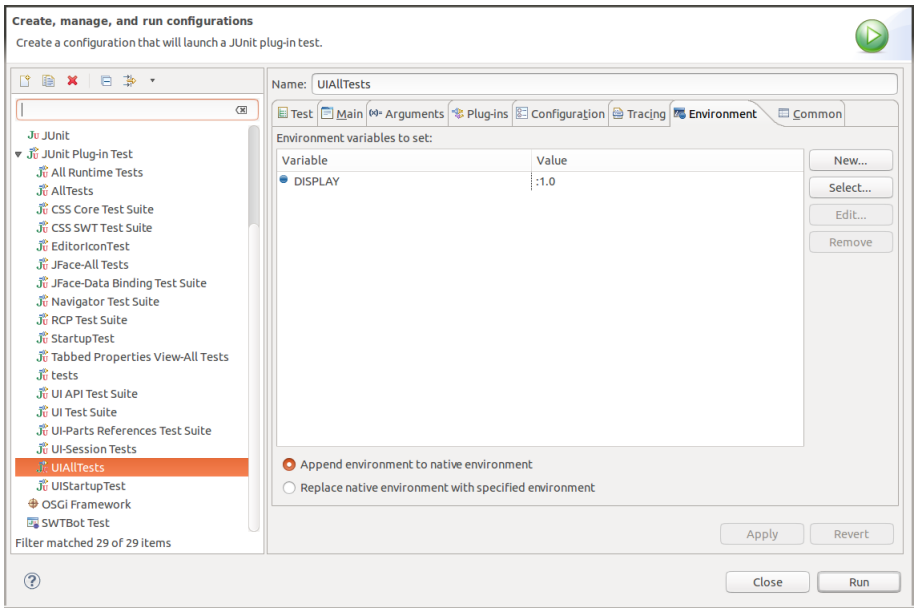
On Ubuntu you can install the virtual server and the client via the following commands. The client is not required to run the unit tests but useful in case you want to view the test execution on the virtual server.

```
# install the server
sudo apt-get install vnc4server

# optionally install the client
sudo apt-get install xtightvncviewer
```

You start the server with the following parameters.

```
# start the server on display #1
vncserver :1 -name testing -depth 24 -geometry 1280x1024
```

In the Eclipse launch configuration you can define the display which is used for the test execution.



If you are running a Maven / Tycho build from the command line, you can export the display variable.

```
export DISPLAY=:1
```

If you want to watch the unit tests, you can also connect to the virtual server via the vncclient.

```
# connect to the server, IP will be asked
vncviewer
```

27
Running the Eclipse platform unit tests

27.1. Repositories for platform tests

Clone the following Git repositories as these contain the existing unit tests. The clone URLs (these are not URLs for the browser!) are listed below.

- git://git.eclipse.org/gitroot/platform/eclipse.platform.releng.git
- git://git.eclipse.org/gitroot/platform/eclipse.platform.ui.git
- git://git.eclipse.org/gitroot/platform/eclipse.platform.runtime.git

In case you want to contribute Gerrit patches to these projects, you should clone them via the Gerrit system as described in Section 13.1, “Finding the correct Git repository”. But you can always adjust the push URL later, as described in Section B.5, “Appendix: Cloning from the Git server and adjusting the push URL”.

27.2. Example: JFace tests

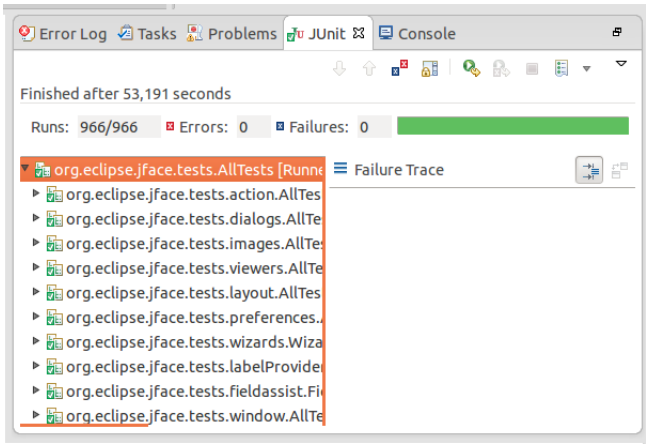
In this section the procedure to run the unit tests of the JFace components is described as an example for running tests from an Eclipse project.

The unit tests for the JFace Bundle are placed in the org.eclipse.ui.tests plug-in. This plug-in includes also several other tests for the Eclipse platform.ui component.

To run theses tests, import the following required plug-ins:

- org.eclipse.core.tests.harness from the eclipse.platform.runtime repository
- org.eclipse.core.tests.runtime from the eclipse.platform.runtime repository
- org.eclipse.test.performance from the eclipse.platform.releng repository
- org.eclipse.ui.tests from the eclipse.platform.ui repository
- org.eclipse.ui.tests.harness from the eclipse.platform.ui repository

To test the full JFace TestSuite, you only need to run the *JFace-All Tests.launch* run configuration from the `org.eclipse.ui.tests` plug-in.


Error Log
Tasks
Problems
JUnit
Console
Finished after 53,191 seconds
Runs: 966/966
Errors: 0
Failures: 0
org.eclipse.jface.tests.AllTests [Runne
Failure Trace
org.eclipse.jface.tests.action.AllTes
org.eclipse.jface.tests.dialogs.AllTe
org.eclipse.jface.tests.images.AllTe
org.eclipse.jface.tests.viewers.AllTe
org.eclipse.jface.tests.layout.AllTes
org.eclipse.jface.tests.preferences.
org.eclipse.jface.tests.wizards.Wiza
org.eclipse.jface.tests.labelProvide
org.eclipse.jface.tests.fieldassist.Fi
org.eclipse.jface.tests.window.AllTe

Part V. Introduction to Eclipse plug-in development

28
Installation for extending the Eclipse IDE

28.1. Java requirements of the Eclipse IDE

For extending the Eclipse IDE, you will need to run Eclipse with at least Java 7 JRE/JDK, as the Mars (Eclipse 4.5) release requires Java 7. As of Neon (Eclipse 4.6) Java 8 is required.

The Eclipse IDE contains its custom Java compiler hence a JRE is sufficient for most tasks with Eclipse. The *JDK* version of Java is only required if you compile Java source code on the command line and for advanced development scenarios. For example, if you use automatic builds or if you develop Java web applications.

Please note that you need a 64 JVM to run a 64 bit Eclipse and a 32 bit JVM to run a 32 bit Eclipse.

28.2. Download the Eclipse SDK

If you plan to add functionalities to the Eclipse platform, you should download the latest Eclipse release. Official releases have stable APIs, therefore are a good foundation for adding your plug-ins and features.

The Eclipse IDE is provided in different flavors. While you can install the necessary tools in any Eclipse package, it is typically easier to download the Eclipse Standard distribution which contains all necessary tools for plug-in development. Other packages adds more tools which are not required for Eclipse plug-in development.

Browse to the *Eclipse download site [http://www.eclipse.org/downloads]* and download the *Eclipse Standard* package.

Note

Eclipse 4.5 provides also a new *https://wiki.eclipse.org/Eclipse_Installer [Eclipse installer]* installer. The installer is useful if you want to download several flavors of Eclipse, as it uses a shared installation pool for common plug-ins.

28.3. Installation the Eclipse IDE

28.3.1. Install the Eclipse IDE

After you downloaded the file with the Eclipse distribution, unpack it to a local directory. Most operating systems can extract zip or tar.gz files in their file browser (e.g., *Windows 7*) with a right-click on the file and selecting "Extract all...".

Note

As a developer person you probably know how to extract a compressed file but if in doubt, search with Google for "How to extract a zip (or tar.gz on Linux and Mac OS) file on ...", replacing "..." with your operating system.

Extract Eclipse into a directory with no spaces in its path, and do not use mapped network drives. Also avoid to have path names longer than 255 characters. Installations of Eclipse in directories with long path names might cause problems, as some Microsoft Windows tooling can not manage these long path names.

After you extracted the compressed file you can start Eclipse, no additional installation procedure is required.

28.3.2. Solving exit code=13 while starting the Eclipse IDE

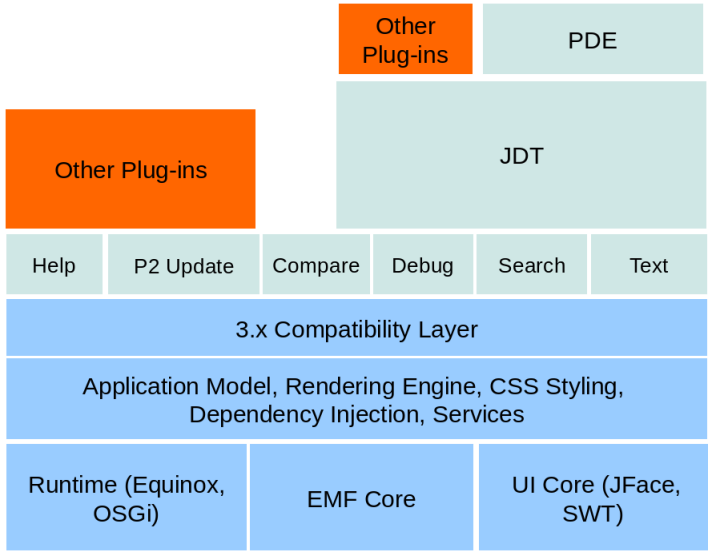
A common startup error happens when the user tries to launch a 64-bit version of Eclipse using a 32 bit JVM or vice versa. In this case Eclipse does not start and the user gets a message containing exit code=13. This happens when the version of Eclipse is not matching the JVM version. A 32-bit Eclipse must run with Java 32 bit, and 64-bit Eclipse must use a 64-bit JVM. Use `java -version` on the command line and if the output does not contain the word "Server" you are using the 32 bit version of Java and must use a 32 bit version of Eclipse.

29
Architecture of Eclipse plug-ins

29.1. Architecture of Eclipse based applications

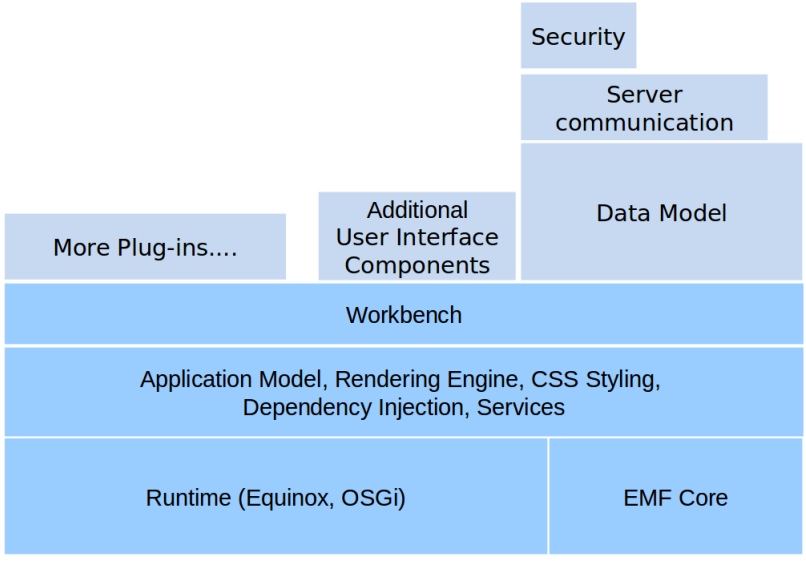
An Eclipse application consists of individual software components. The Eclipse IDE can be viewed as a special Eclipse application with the focus on supporting software development.

The core components of the Eclipse IDE are depicted in the following graphic. The intention of the graphic is to demonstrate the general concept, the displayed relationships are not 100 % accurate.



The most important Eclipse components of this graphic are described in the next section. On top of these base components, the Eclipse IDE adds additional components which are important for an IDE application, for example, the Java development tools (JDT) or version control support (EGit).

An Eclipse RCP application typically uses the same base components of the Eclipse platform and adds additional application specific components as depicted in the following graphic.



29.2. Core components of the Eclipse platform

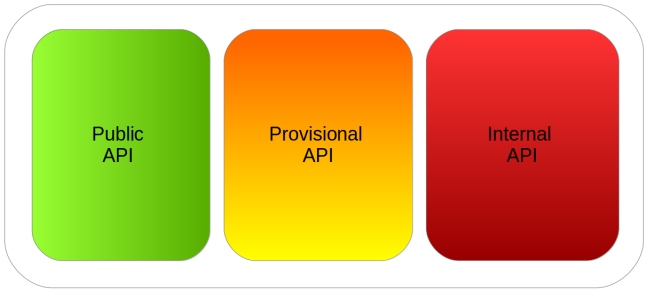
OSGi is a specification to describe a modular approach to develop component-based Java applications. The programming model of OSGi allows you to define dynamic software components, i.e., OSGi services. *Equinox* is one implementation of the OSGi specification and is used by the Eclipse platform as its runtime. This Equinox runtime provides the necessary API's and framework to run a modular Eclipse application.

SWT is the standard user interface component library used by Eclipse. *JFace* provides some convenient APIs on top of SWT. The *workbench* provides the framework for the application. It is responsible for displaying all other user interface components.

EMF is the Eclipse Modeling Framework which provides functionality to model a data model and to use this data model at runtime.

29.3. Eclipse API and internal API

An OSGi runtime allows the developer to mark Java packages as public, provisional or internal APIs. The internal API is private, therefore not visible. The provisional API are to test non-finalized APIs, therefore are visible but non-stable. The public API, or simply API, are the visible and stable API, that can be reused by other components.



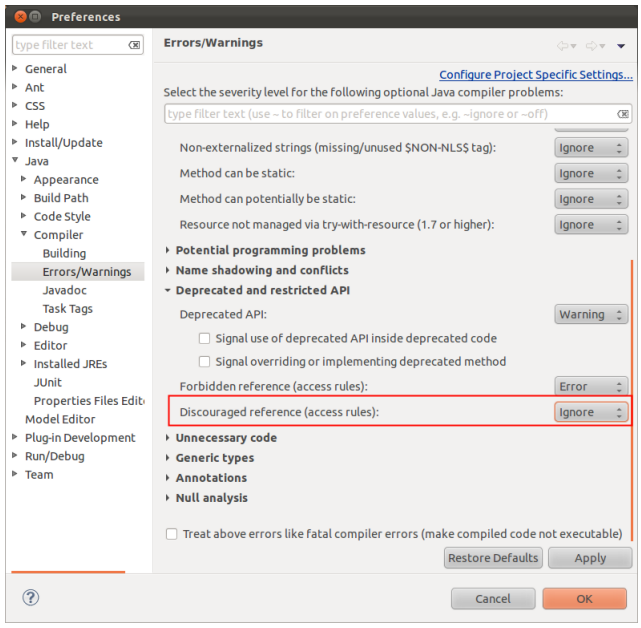
The Eclipse platform project marks packages either as public API or as provisional API, to make all Java classes accessible to Eclipse developers. If the Eclipse platform project releases an API, the platform project plans to keep this API stable for as long as possible.

If API is internal but accessible, i.e., marked as provisional, the platform team can change this API in the future. If you use such API, you must be prepared that you might have to make some adjustments to your application in a future Eclipse release.

If you use unreleased API, you see a *Discouraged access: The ...is not API (restriction on required project ...)* warning in the Java editor.

Tip

You can turn off these warnings for your workspace via *Window* → *Preferences* → *Java* → *Compiler* → *Errors/Warnings* and by setting the *Discouraged reference (access rules)* flag to *Ignore*.



Alternatively you can turn off these warnings on a per project basis, via right-click on the project *Properties* → *Java Compiler* and afterwards use the same path as for accessing the global settings. You might have to activate the *Enable project specific settings* checkbox at the top of the Error/Warnings preference page.

29.4. Important configuration files for Eclipse plug-ins

An Eclipse plug-in has the following main configuration files. These files are defining the API, and the dependencies of the plug-in.

- `MANIFEST.MF` - contains the OSGi configuration information.
- `plugin.xml` - optional configuration file, contains information about Eclipse specific extension mechanisms.

An Eclipse plug-in defines its meta data, like its unique identifier, its exported API and its dependencies via the `MANIFEST.MF` file.

The `plugin.xml` file provides the possibility to create and contribute to Eclipse specific API. You can add *extension points* and *extensions* in this file. *Extension-points* define interfaces for other plug-ins to contribute functionality. *Extensions* contribute functionality to these interfaces. Functionality can be code and non-code based. For example a plug-in might contain help content.

30
Starting an Eclipse instance (Runtime Eclipse)

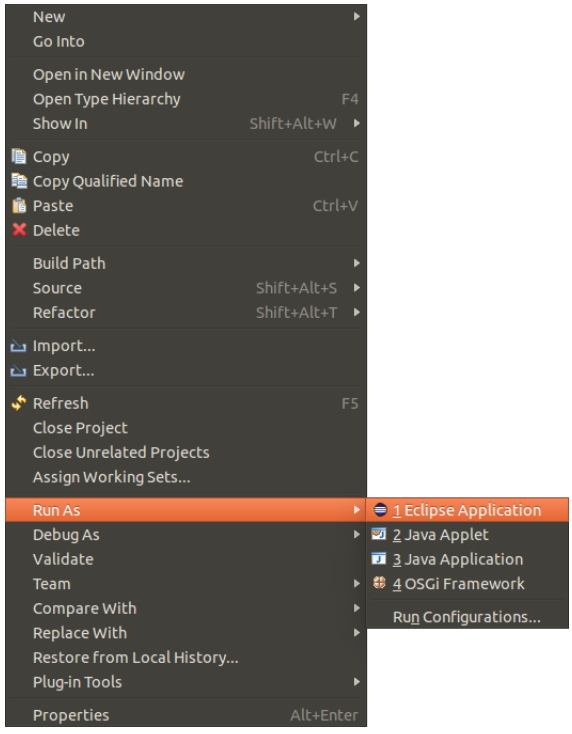
30.1. Starting the Eclipse IDE from Eclipse

During the development of plug-ins targeting the Eclipse IDE you frequently have to deploy your plug-ins into a new Eclipse IDE instance to test your development.

As this is a very common requirement, it is possible to start an instance of the Eclipse IDE from another Eclipse IDE instance. This is sometimes called a *runtime Eclipse IDE*. For this you specify the set of plug-ins which should be included in the Eclispe instance via a run Configuration.

30.2. Starting a new Eclipse instance

The easiest way to create a run configuration for an Eclipse IDE is by selecting *Run As* → *Eclipse Application* from the context menu of an existing plug-in or your manifest file. By default, this takes all the plug-ins from the workspace and the target environment, and start an Eclipse IDE instance with these plug-ins. If a plug-in is available in the workspace and the target environment, the one from the workspace is used.



30.3. Debugging the Eclipse instance

You can debug the Eclipse IDE itself. Put a breakpoint in a line of the source code which you want to inspect, right-click your plug-in and select *Debug As* → *Eclipse Application*. This creates a run configuration if it does not already exist.

When the flow will reach a statement marked with a breakpoint, the execution stops, and you are able to debug the related statement, and to inspect the current data.

31
The usage of run configurations

31.1. What are run configurations?

A *run configuration* defines the environment which will be used to execute a generic launch. For example, it defines arguments to the Java virtual machine (VM), plug-in (classpath) dependencies, etc.

If you start an Eclipse application the corresponding run configuration is automatically created or updated.

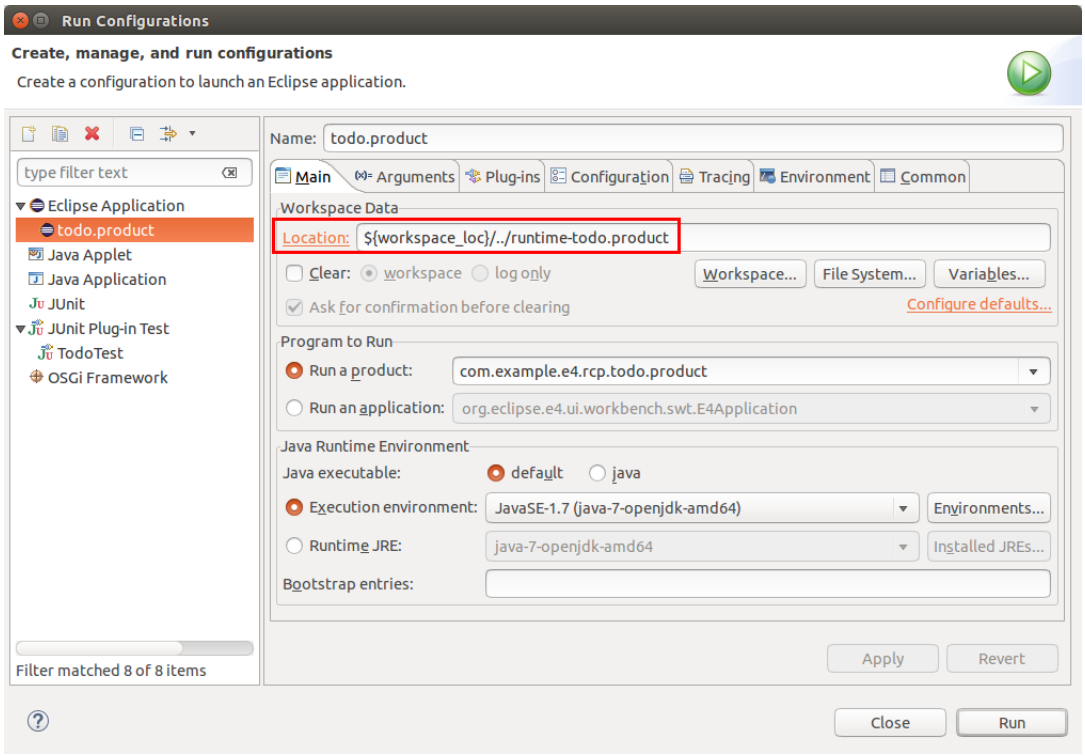
Tip

If you are starting an Eclipse runtime IDE for testing, and clear the workspace location the next run will prompt you to choose a workspace.

31.2. Reviewing run configurations

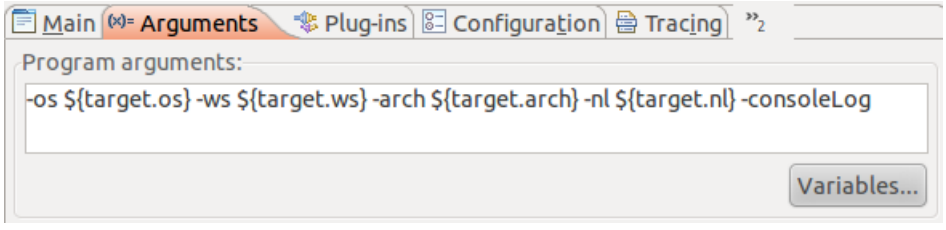
To review and edit your run configurations select *Run* → *Run Configurations...* from the Eclipse menu.

On the *Main* tab in the field *Location*, you specify where the Eclipse IDE creates the files necessary to start your Eclipse based application.



31.3. Run arguments

The run configuration allows you to add additional start arguments for your application on the *Arguments* tab. By default Eclipse includes already several parameters, e.g. parameters for `-os`, `-ws` and `-arch` to specify the architecture on which the application is running.



Tip

To pass system properties to your Eclipse application, you can add launch parameters using the -D switch. As example, if you add the argument `-Dmy.product.loglevel=INFO`, you can get the "INFO" value with `System.getProperties("my.product.loglevel")`.

The following table lists several useful launch arguments.

Table 31.1. Launch parameters

Parameter	Description
`consoleLog`	Error messages of the running Eclipse application are written to standard-out (`System.out`) which can be viewed in the Eclipse IDE *Console* view that started the RCP application.
`nl`	Specifies the locale used for your application. The locale defines the language specific settings, i.e., which translation is used and the number, date and currency formatting. For example `-nl en` starts your application using the English language. This is useful for testing translations.
`console`	Provides access to an OSGi console where you can check the status of your application.
`noExit`	Keeps the OSGi console open even if the application crashes. This allows to analyze the application dependencies even if the application crashes during startup.
`clearPersistedState`	Deletes cached runtime changes of the Eclipse 4 application model.

32
Adding e4 commands, menus and toolbars to a 3.x API based applications

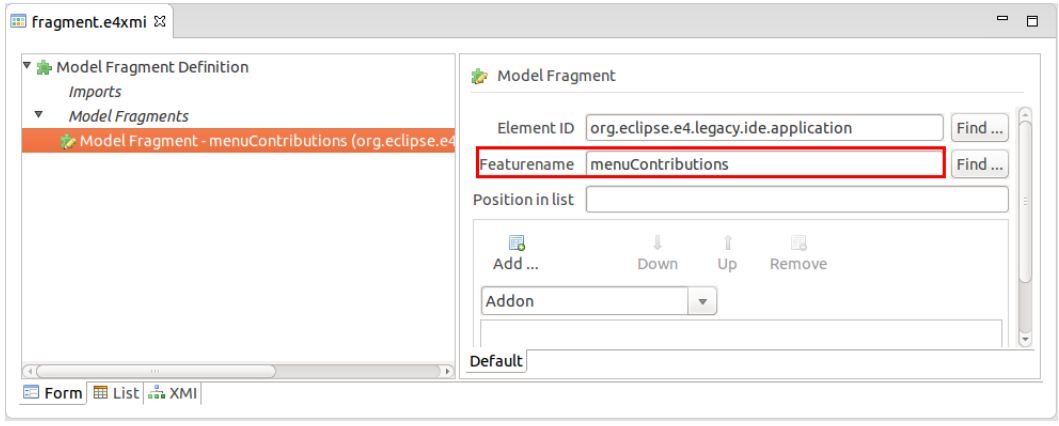
32.1. Adding e4 menu entries

Menus, handlers and commands can be contributed to an Eclipse application via model fragments. To do this, you only need to know the ID of the element to which you want to contribute. The ID of the Eclipse IDE and Eclipse 3.x RCP applications is hard coded to the `org.eclipse.e4.legacy.ide.application` value.

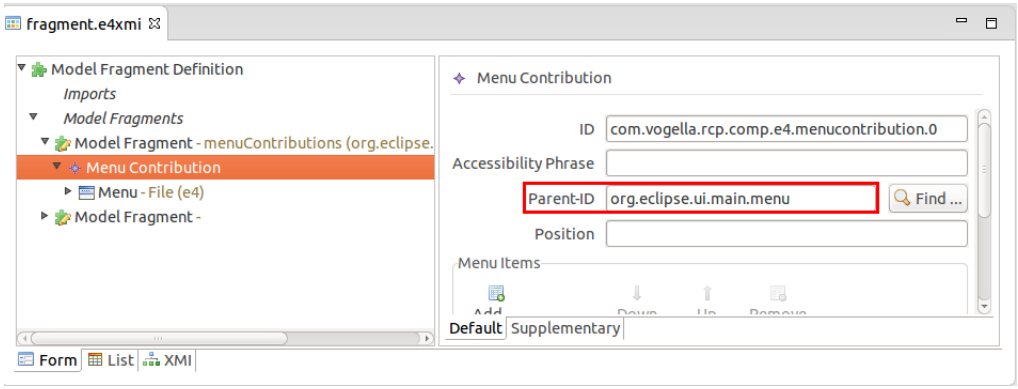
Tip

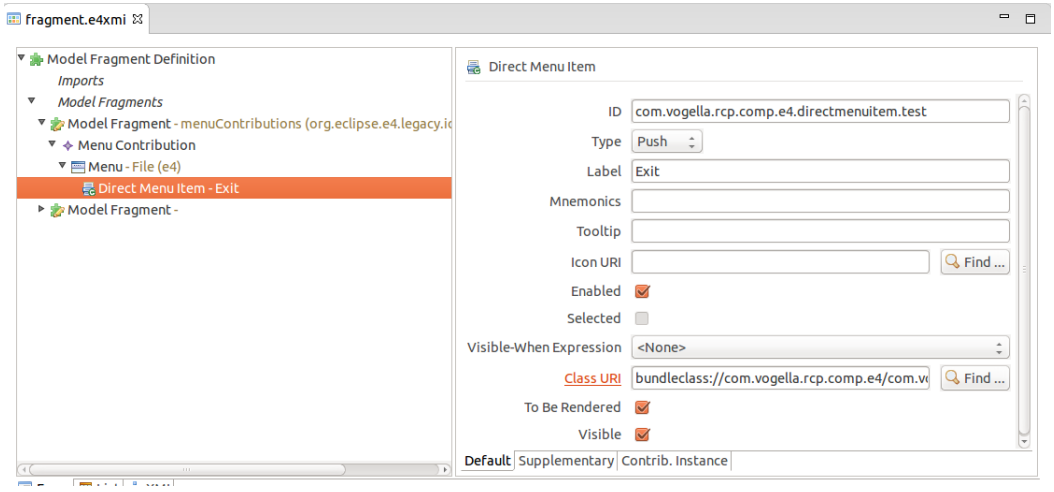
You can use the model spy from the e4 tools project to identify the ID of the element you want to contribute too. See Section 40.3, “Analyzing the application model with the model spy”.

With the correct ID you can create model fragments that contribute to the corresponding application model element. The following screenshot demonstrate how to contribute to the *menuContributions* feature of the Eclipse IDE.



After you added a MenuContribution item you can contribute a menu. The `Parent-ID` must be the ID of the menu your are contributing to.





The model fragment must be registered in the plugin.xml file via an extension to the `org.eclipse.e4.workbench.model` extension point, as demonstrated in the following listing.

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin>
   <extension
         id="id1"
         point="org.eclipse.e4.workbench.model">
      <fragment
            apply="notexists"
            uri="fragment.e4xmi">
      </fragment>
   </extension>

</plugin>
```

32.2. Error analysis

In case of problems, check the source code of plugin.xml and validate that your model fragment is included. Verify the name of the referred model fragment and ensure all the defined ID are available in the running configuration.

32.3. Adding e4 toolbar entries to the appliation window

Similar to menus you can contribute toolbar contributions. This is demonstrated in Section 33.7, “Adding a toolbar contribution”.

Note

This approach does currently not work for view toolbars.

33
Exercise: Add a e4 menu and toolbar to the Eclipse IDE

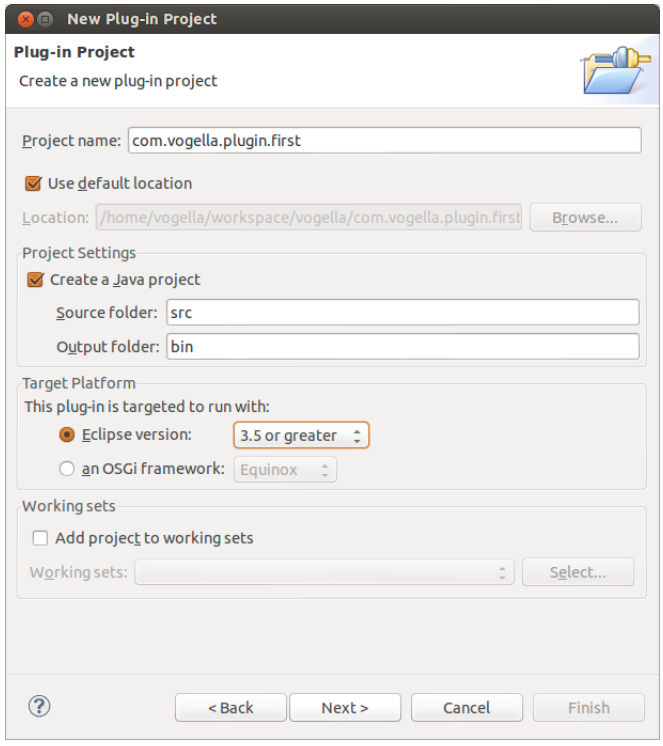
33.1. Target of this exercise

In this exercise you create a plug-in which contributes an e4 menu entry to a 3.x based application menu.

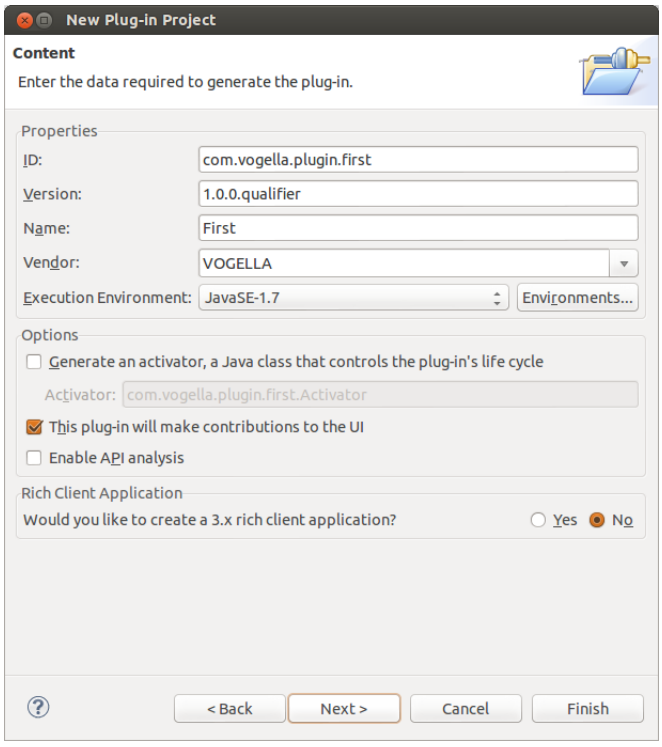
33.2. Creating a plug-in project

Create a new plug-in project called `com.vogella.plugin.first` via *File* → *New* → *Project...* → *Plug-in Development* → *Plug-in Project*.

Enter the data as depicted in the following screenshots.

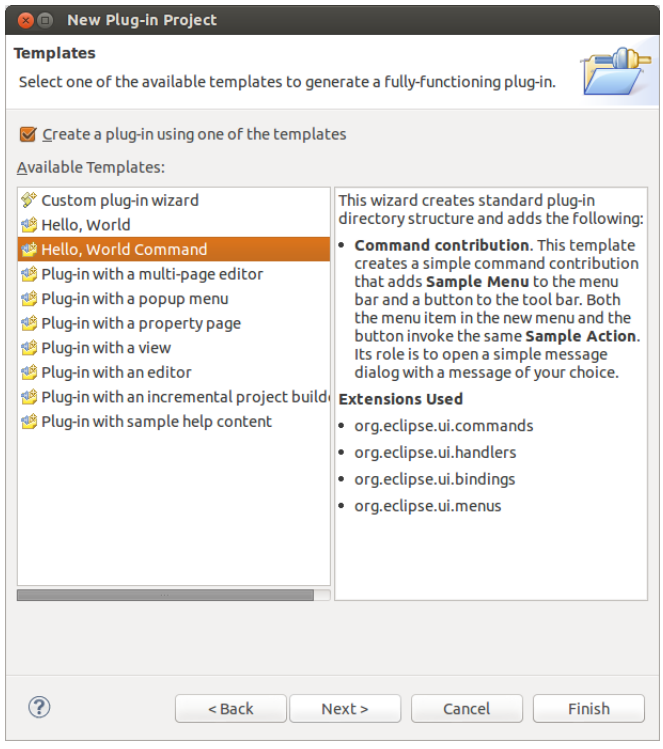


Press the *Next* button.

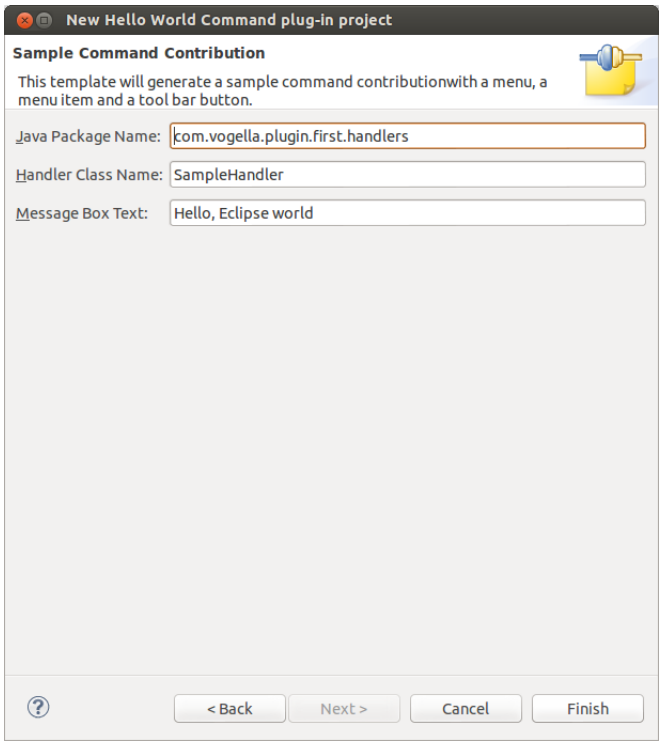


Press the *Next* button.

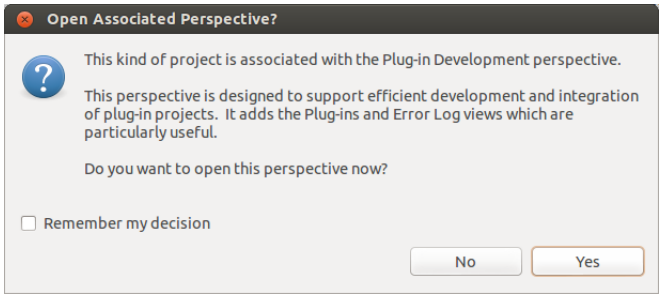
Select the *Hello, World Command* template and press the *Next* button. This template uses the 3.x API, which you convert to the e4 API in Section 33.6, “Creating a model contribution”.



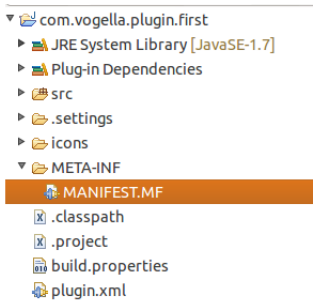
In the last page of the wizard you can customize some suggested values. You can leave the default values and press the *Finish* button.



Eclipse may ask you if you want to switch to the plug-in development perspective. Answer *Yes* if you are prompted.



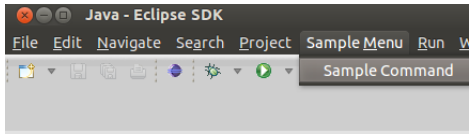
As a result the following project is created.

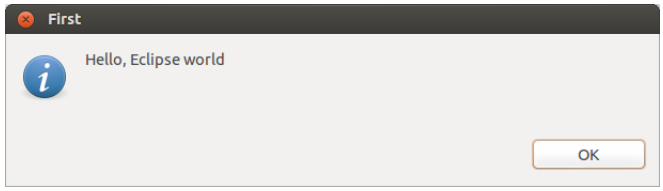


33.3. Starting an Eclipse runtime IDE with your plug-in

Start a new Eclipse IDE instance and validate that your menu and toolbar entry are available. See Section 30.2, “Starting a new Eclipse instance” for more information on how to start an instance of the Eclipse IDE with additional plug-ins.

After launching the Eclipse IDE instance, you see the sample menu, contributed by your plug-in. After clicking the menu entry, you see a message box.





33.4. Adding the plug-in dependencies for the e4 API

Add a dependency to the `org.eclipse.e4.core.di` plug-in in the manifest file of the newly created plug-in.

33.5. Creating the handler class

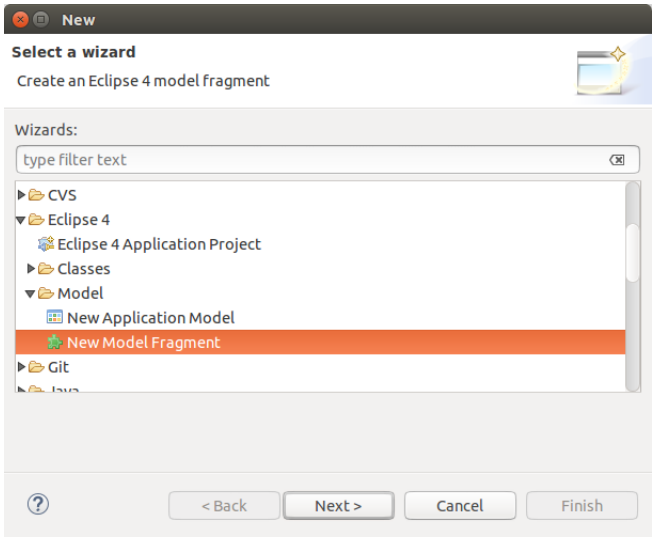
Create the following class based on the generated handler class.

```
package com.vogella.plugin.first.handlers;

import org.eclipse.e4.core.di.annotations.Execute;
import org.eclipse.jface.dialogs.MessageDialog;
import org.eclipse.swt.widgets.Shell;

public class SampleE4Handler {
```

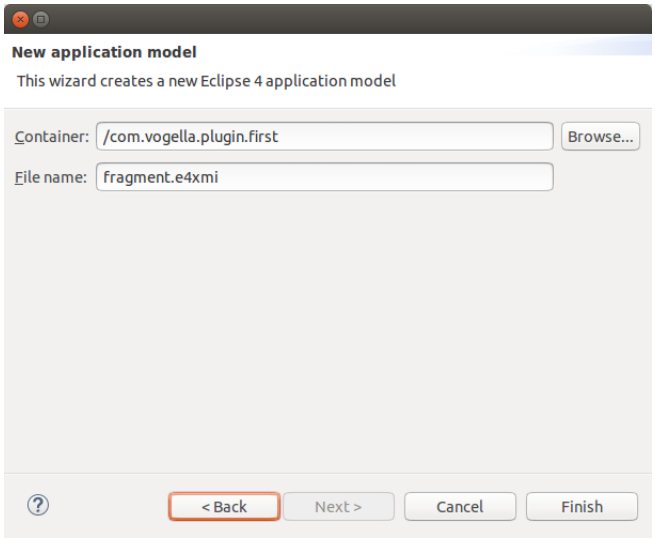
```
  @Execute
  public void execute(Shell shell) {
    MessageDialog.openInformation(shell, "First", "Hello, e4 API world");
  }
}
```

33.6. Creating a model contribution

Select *New* → *Other...* → *Eclipse 4* → *Model* → *New Model Fragment* from the context menu of the plug-in project.

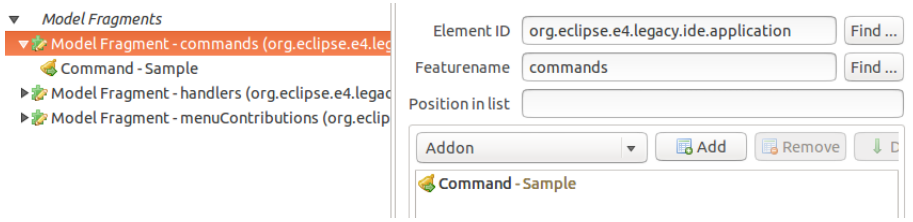


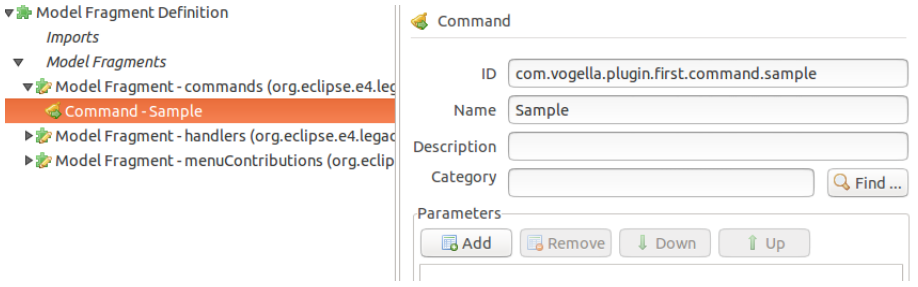
Press the *Finish* button.

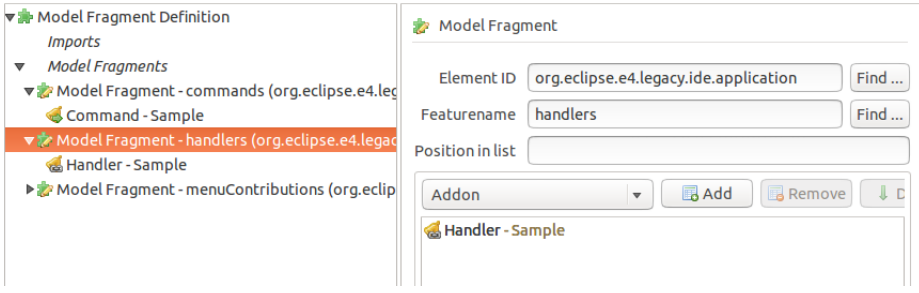


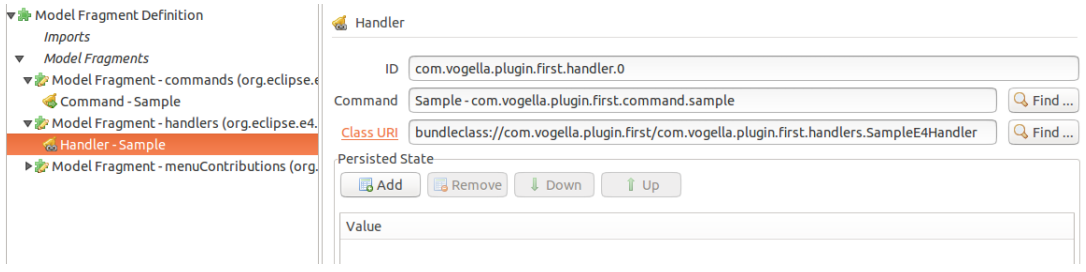
Create three model fragment entries in your new file, all of them should be contributing to the `org.eclipse.e4.legacy.ide.application` element id.

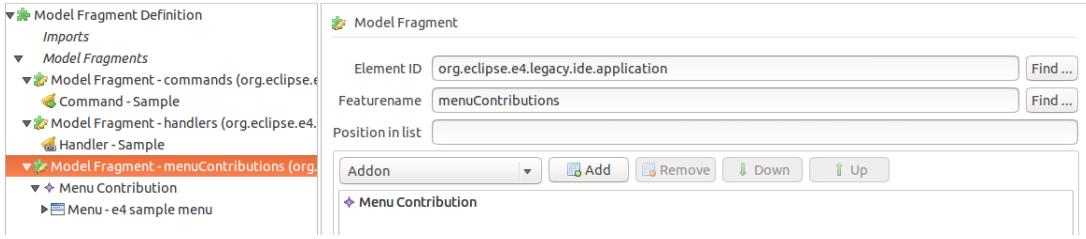
Use the following screenshots to define the new contribution.

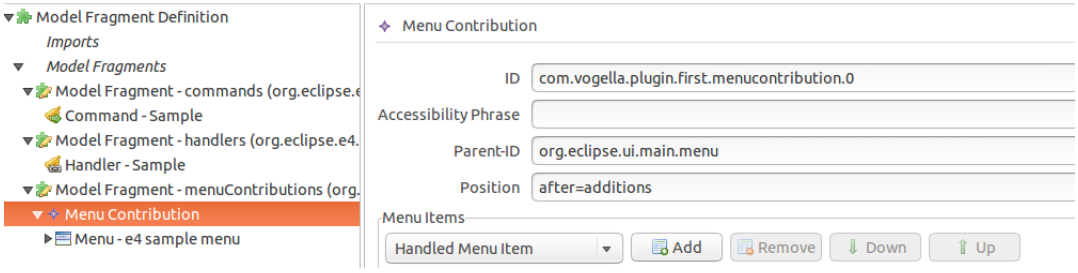


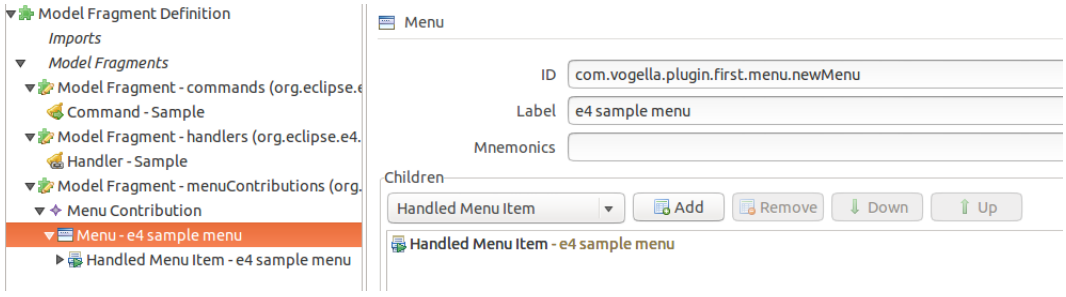


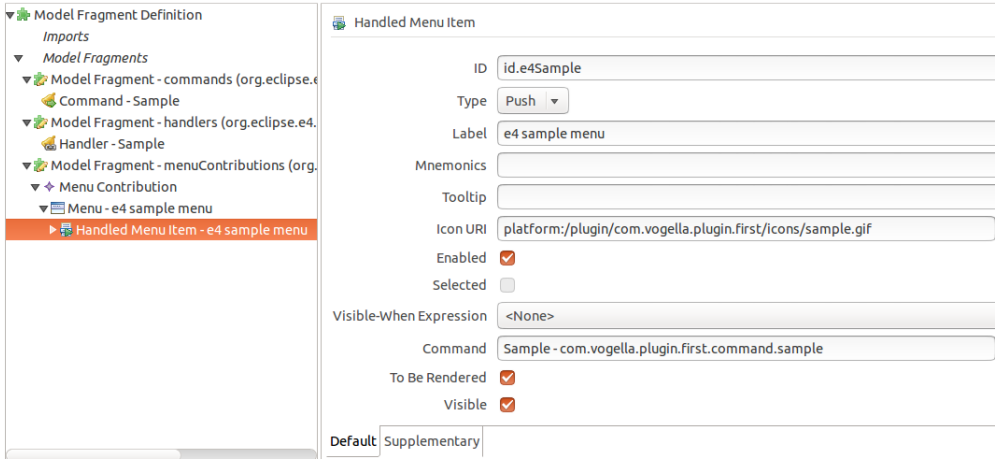






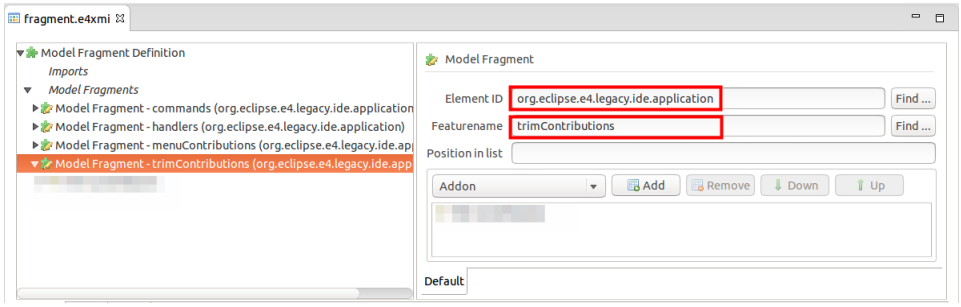






33.7. Adding a toolbar contribution

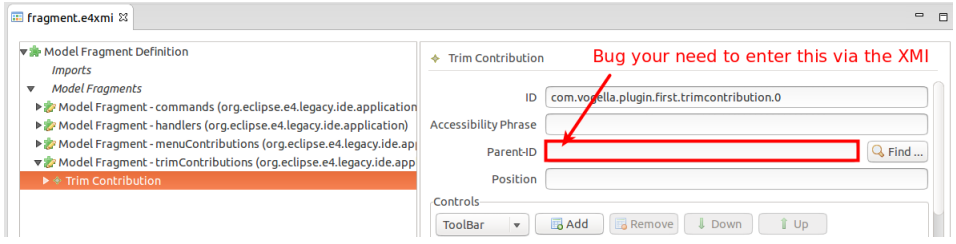
Also add a toolbar contribution for the same command.



Warning

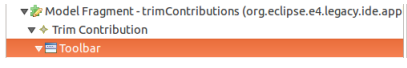
When this document was written, the model editor had a bug. If you enter a *Parent-ID* to the toolbar contribution, that information is not persisted in the xmi code. Therefore, ensure that your settings are actually reflected in the fragment file. You can do this by closing the file and opening it again.

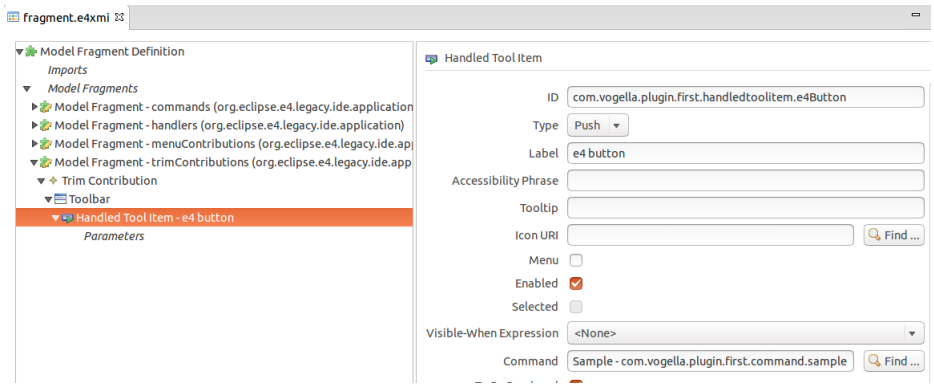
If you read this and the bug is still present, you have to enter the Parent-ID directly in the xmi. For this switch to the XMI tab of the editor and enter the parent ID directly. Use the `org.eclipse.ui.main.toolbar` value. The file is only saved, if you enter the information syntactically correct.




```
<fragments xsi:type="fragment:StringModelFragment" xmi:id="_xr
  <elements xsi:type="menu:TrimContribution"
      xmi:id=" 4LRJcMaBEeSYi86KPuHLFg"
      elementId="com.vogella.plugin.first.trimcontribution.0"
      parentId="org.eclipse.ui.main.toolbar">
```

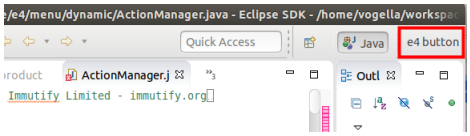
Add a toolbar and a handled tool item to your contribution.





33.8. Validating the presence of the menu and toolbar contribution

Start a new instance of the Eclipse IDE and validate that your menu and the toolbar are is contributed. If they are not visible in the window, check via the model spy for potential issues.



34
Features and feature projects

34.1. What are feature projects and features?

Using the Eclipse wizards you can create feature projects, similarly on how you create plug-in projects. An Eclipse feature project contains *features*.

A feature describes a list of plug-ins and other features which can be seen as a logical unit, i.e., a set of related components. It also has a name, a version number and in most cases a license information assigned to it.

A feature is described via a `feature.xml` file. The following listing shows an example of such a file.

```
<?xml version="1.0" encoding="UTF-8"?>
<feature
      id="com.vogella.eclipse.featuretest.feature"
      label="Feature"
      version="1.0.0.qualifier"
      provider-name="vogella GmbH">

   <description url="http://www.example.com/description">
      [Enter Feature Description here.]
   </description>

   <copyright url="http://www.example.com/copyright">
      [Enter Copyright Description here.]
   </copyright>

   <license url="http://www.example.com/license">
      [Enter License Description here.]
   </license>

   <plugin
         id="com.vogella.eclipse.featuretest.feature"
         download-size="0"
         install-size="0"
         version="0.0.0"
         unpack="false"/>

</feature>
```

34.2. Creating a feature

You can create a new feature project via the following menu path: *File* → *New* → *Other...* → *Plug-in Development* → *Feature Project*.

34.3. The purpose of the tabs in the feature editor

If you open the `feature.xml` file you can change the feature properties via a special editor.

The *Information* tab allows you to enter a description and copyright related information for this feature.

The *Included Plug-ins* tab allows you to change the included plug-ins in the feature. If you want to add a plug-in to a feature, use this tab. A frequent error of new Eclipse developers is to add it to the *Dependencies* tab.

The *Included Features* tab allows you to include other features into this feature. Via the *Dependencies* tab you can define other features which must be present to use this feature.

The *Build* tab is used for the build process and should include the `feature.xml` file. The last two tabs give access to the configuration files in text format.

34.4. Advantages of using features

The grouping of plug-ins into logical units makes it easier to handle a set of plug-ins. Instead of adding many individual plug-ins to your product configuration file, you can group them using features. That increases the visibility of your application structure.

Features can be consumed by the Eclipse update manager, the build process and optionally for the definition of Eclipse products. Features can also be used as basis to define a launch configuration.

Eclipse provides several predefined features, e.g. the `org.eclipse.e4.rcp` for Eclipse RCP applications.

35
Deploy your software component (plug-ins) locally

35.1. Options for installing a plug-in

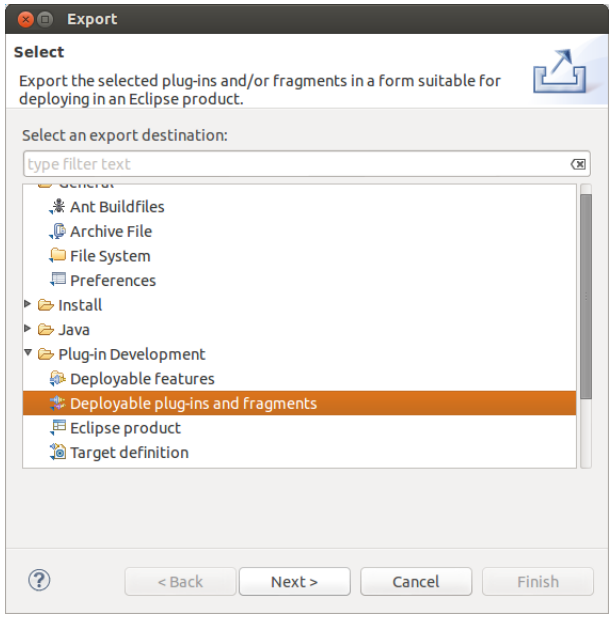
You have several options to make your plug-in available in your Eclipse IDE. You can:

- Install your plug-in directly into your current Eclipse installation, using your Eclipse IDE. This option is useful for a quick installation and a test during development.
- Export your plug-in and copy it into the `dropins` folder of your Eclipse installation. This option is useful for a simple distribution scenario, where you only have to provide the exported plug-in to your users. On the other hand it is not very convenient for the end user as it requires manual file copying and provides no simple way to update the plug-in.
- Create an update site and use the Eclipse update manager to install it from this site. This is the simplest approach for the end user. It also allows providing updates to all interested users.

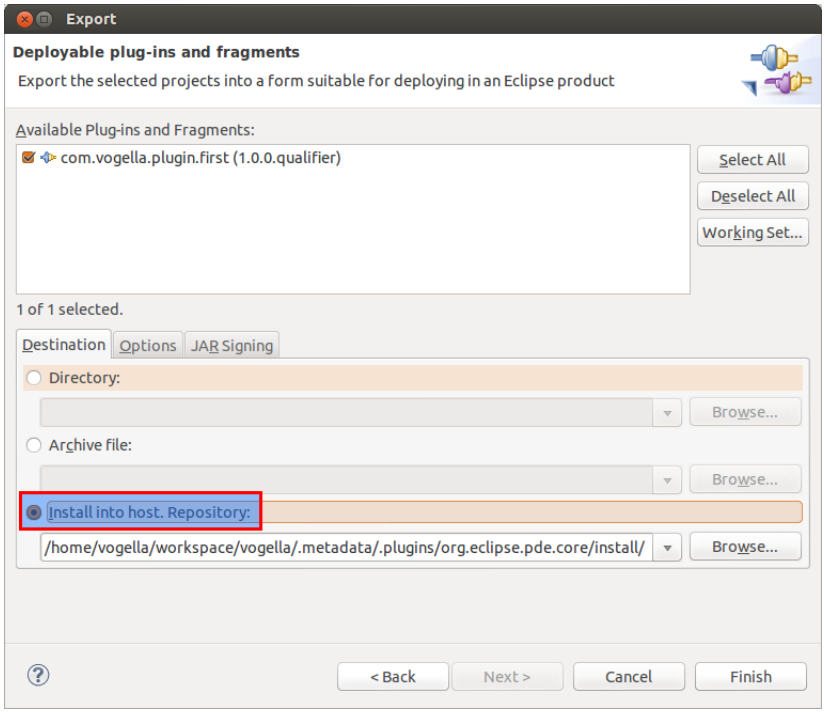
35.2. Installing your plug-in from your Eclipse IDE

You can install your plug-in directly into your running Eclipse IDE.

The Eclipse plug-in export wizard has an option for this. Open the export wizard via *File* → *Export* → *Plug-in Development* → *Deployable plug-ins and fragments*.



In the export wizard dialog select in this case *Install into host. Repository*. This is depicted in the following screenshot.

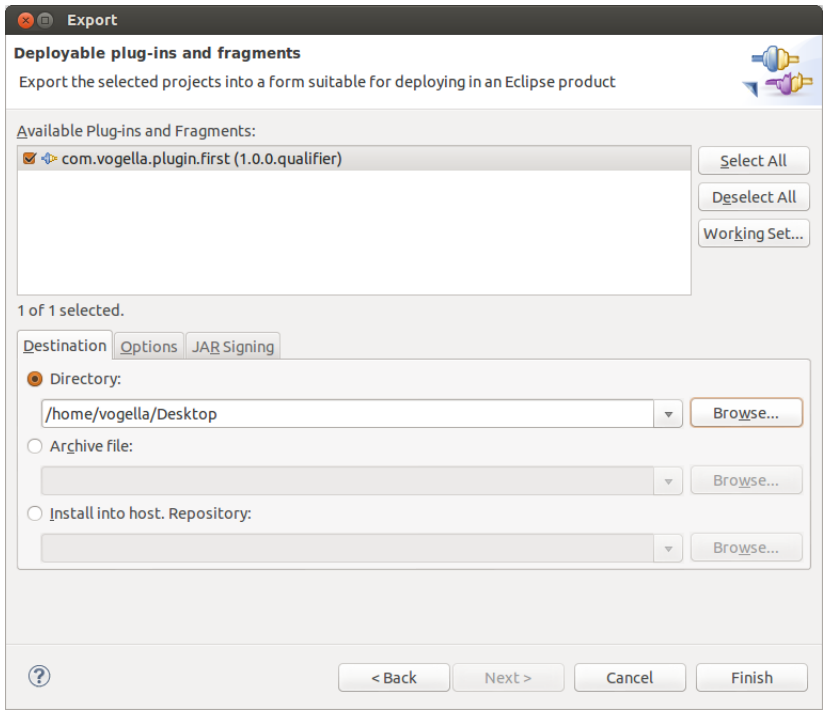


35.3. Export plug-in and put into dropins folder

If you export your plug-in locally, you can put it into the Eclipse `dropins` folder of your Eclipse installation. After a restart of your Eclipse your plug-in should be available and ready for use.

Open again the export wizard via *File → Export → Plug-in Development → Deployable plug-ins and fragments*.

Select the plug-in you want to export and the folder to which this plug-in should get exported.



Press the *Finish* button. This creates a `JAR` file with the exported plug-in in the selected directory.

Copy this `JAR` file to the `dropins` directory in your Eclipse installation directory and restart your running Eclipse.

After this restart your new plug-in is available in your Eclipse installation and ready to be used.

35.4. Deployment using an update site

You can create an update site for your plug-in. An update site consists of static files, which can be placed on a file server or a web server. Other users can install Eclipse plug-in from this update site by providing them a link to the update site.

This requires that you create a feature project for the plug-in. You can export this feature project and use the Eclipse update manager to install the feature (with the plug-in).

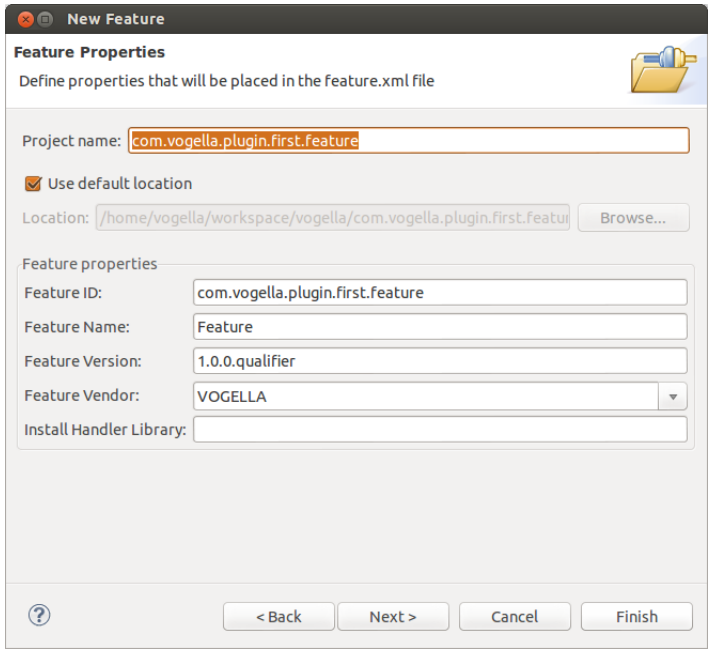
By default, the Eclipse update manager shows only features included into a category. Therefore, you should always use a category for your exported feature to make it easy for the user to install your feature.

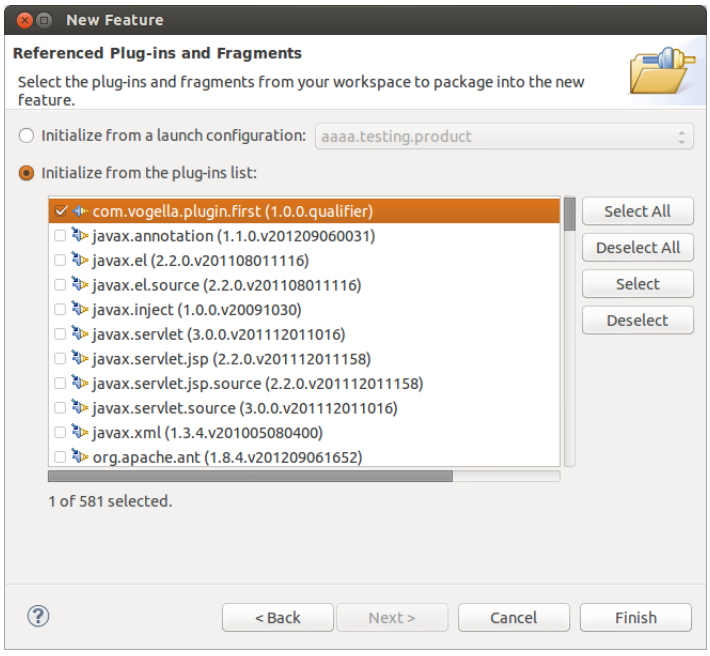
36
Exercise: Create a feature for your plug-in

36.1. Create a feature project

Create a feature project for your plug-in and add your plug-in to this feature. You create a feature project via *File* → *New* → *Other...* → *Plug-in Development* → *Feature Project*.

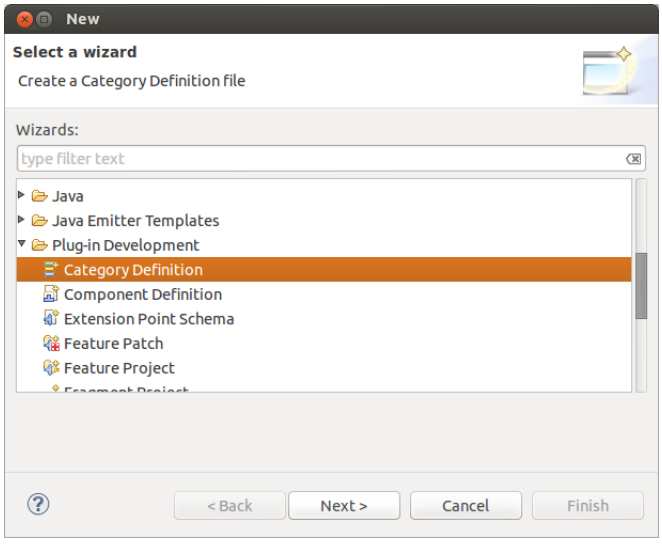
Create the feature project according to the following screenshots.



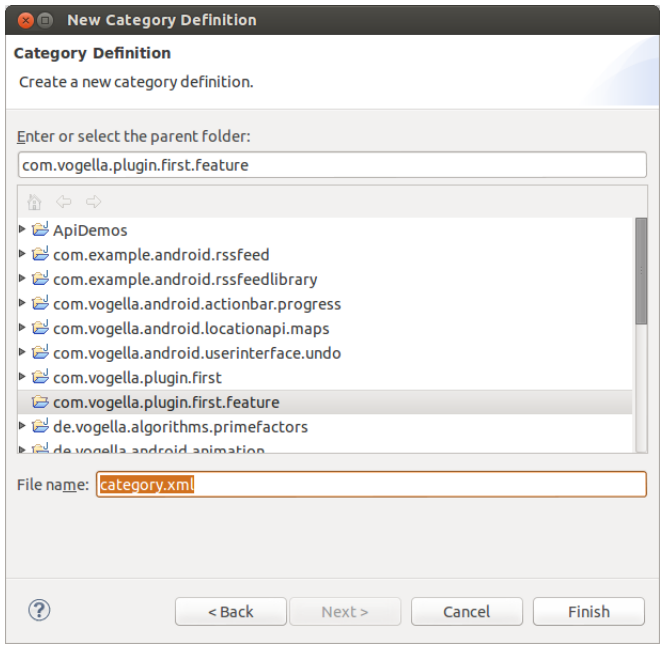


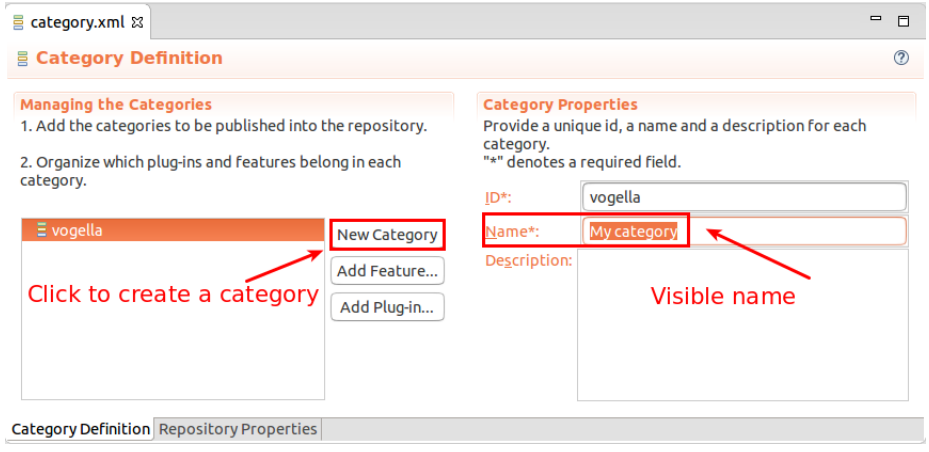
36.2. Create a category definition

In your feature project create a new category definition, via the menu entry *File* → *New* → *Other...* → *Plug-in development* → *Category Definition*.



Press the *New Category* button and create a category with a name which describes your functionality. Add your feature to this category.

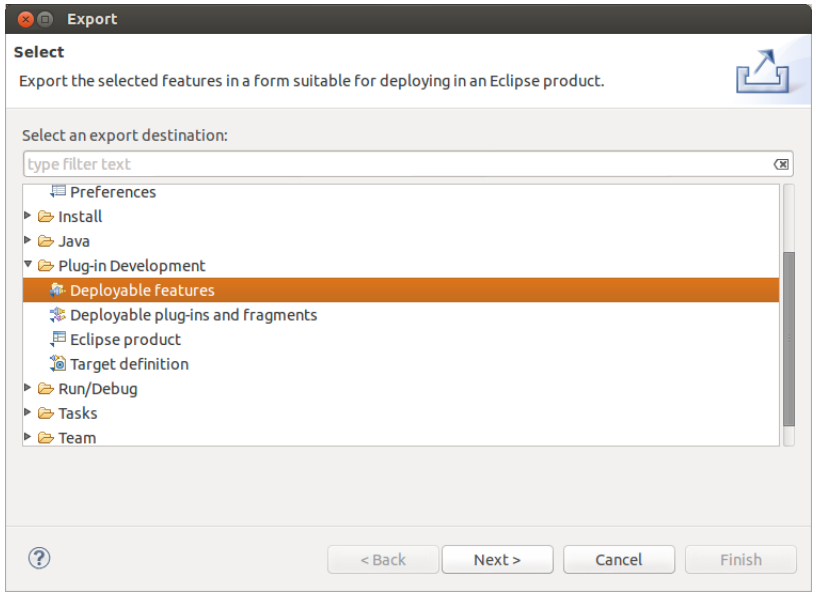

New Category Definition
Category Definition
Create a new category definition.
Enter or select the parent folder:
com.vogella.plugin.first.feature
ApiDemos
com.example.android.rssfeed
com.example.android.rssfeedlibrary
com.vogella.android.actionbar.progress
com.vogella.android.locationapi.maps
com.vogella.android.userinterface.undo
com.vogella.plugin.first
com.vogella.plugin.first.feature
de.vogella.algorithms.primefactors
File name:
category.xml
< Back
Next >
Cancel
Finish

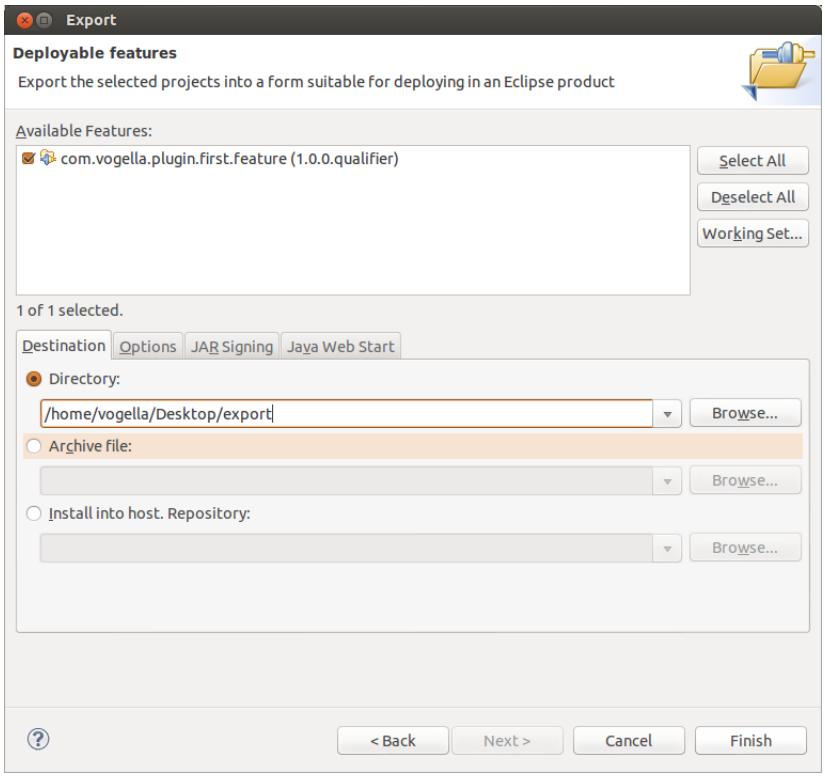

category.xml
Category Definition
Managing the Categories
1. Add the categories to be published into the repository.
2. Organize which plug-ins and features belong in each category.
vogella
New Category
Add Feature...
Add Plug-in...
Click to create a category
Category Properties
Provide a unique id, a name and a description for each category.
"*" denotes a required field.
ID*:
vogella
Name*:
My category
Description:
Visible name
Category Definition
Repository Properties

37
Exercise: Create update site and install plug-in from it

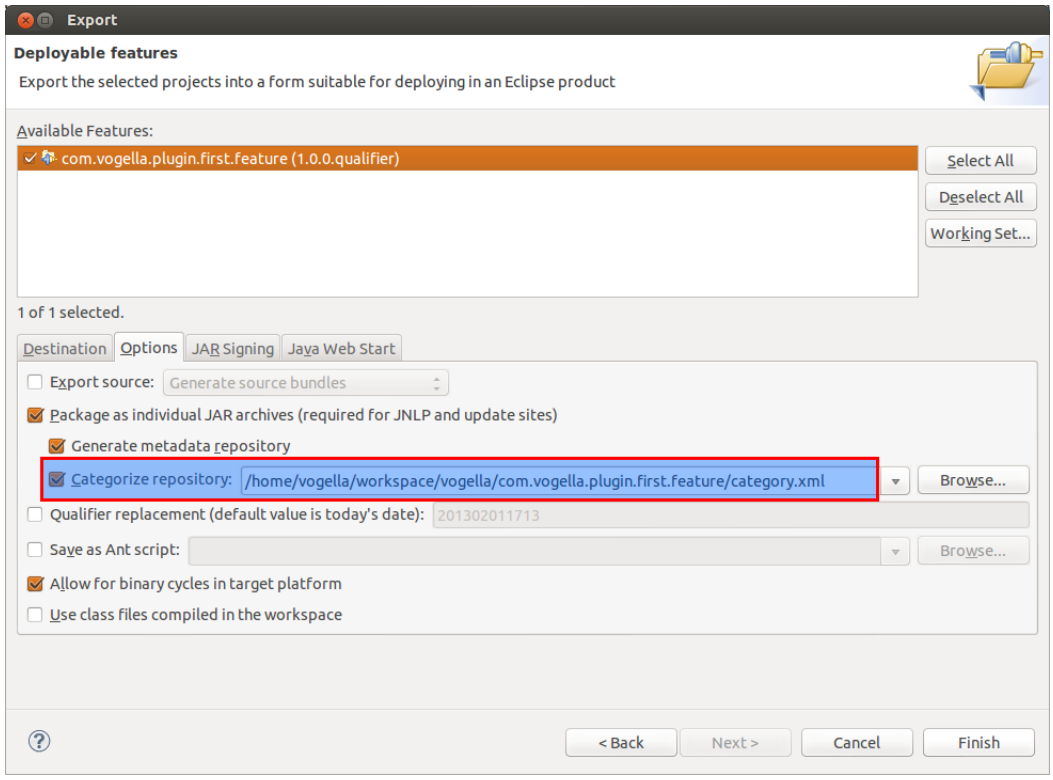
37.1. Create an update site

You can create an update site for your feature in a local directory on your machine. For this, select *File* → *Export* → *Deployable features*.





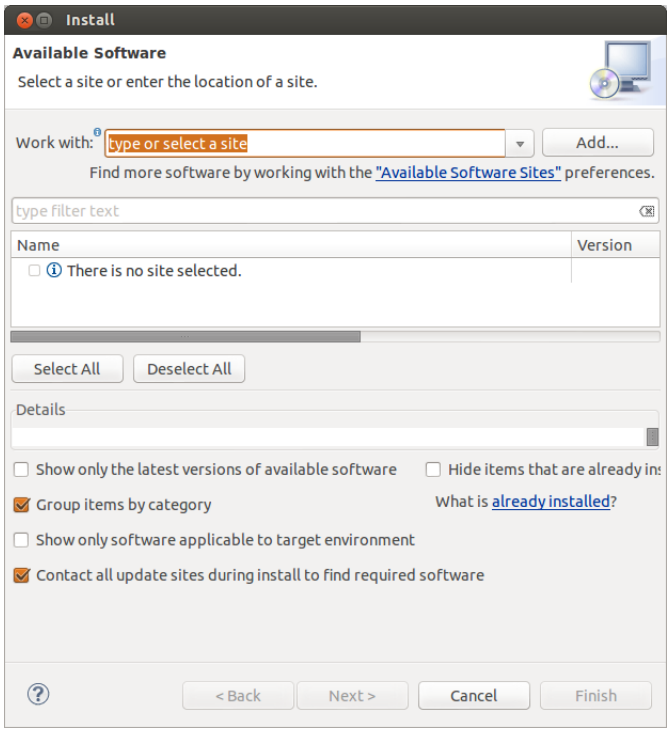
To use your category, switch to the *Options* tab and select the path to your `category.xml` file in the *Categorize repository* option.

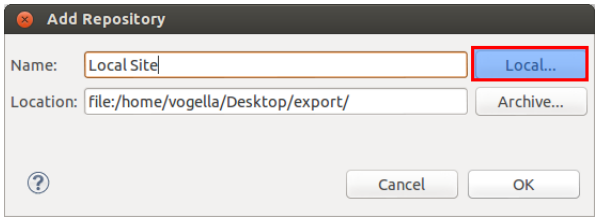


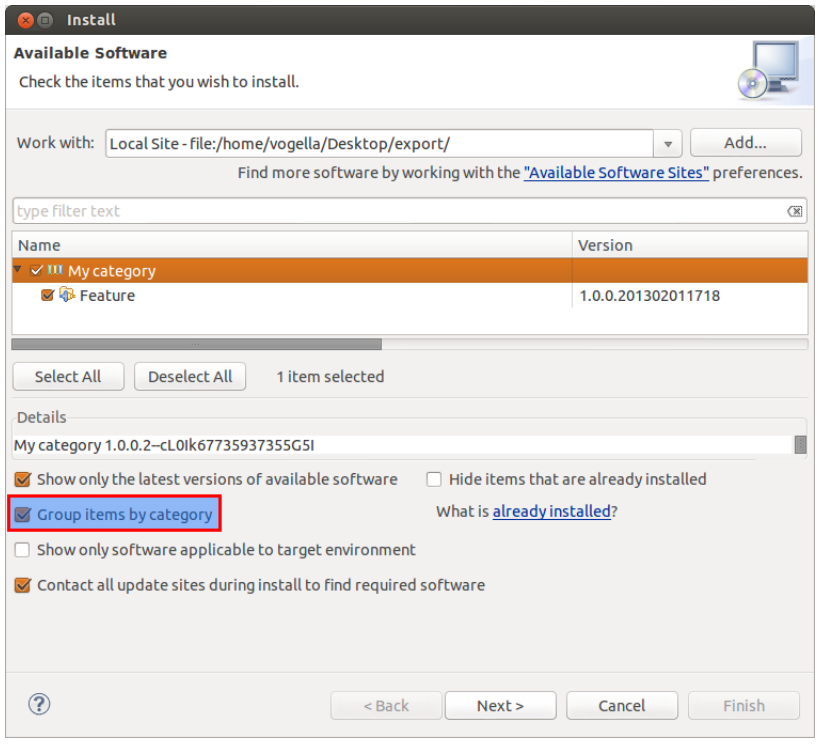
37.2. Install feature via the Eclipse update manager

Use the Eclipse update manager via *Help → Install New Software...* to install this new feature into your Eclipse IDE.

Use the update manager to point to your local directory. Then select and install your feature. In case you don't see your feature, try deselecting the *Group items by category* flag. In this case, you have forgotten to set your category during the export.


Install
Available Software
Select a site or enter the location of a site.
Work with:
type or select a site
Add...
Find more software by working with the "Available Software Sites" preferences.
type filter text
Name
Version
There is no site selected.
Select All
Deselect All
Details
Show only the latest versions of available software
Hide items that are already ins
Group items by category
What is already installed?
Show only software applicable to target environment
Contact all update sites during install to find required software
< Back
Next >
Cancel
Finish


Add Repository
Name:
Local Site
Local...
Location:
file:/home/vogella/Desktop/export/
Archive...
Cancel
OK

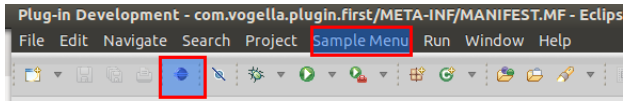


Note

If you put the resulting files on a webserver under a public accessible URL, your users could install your features from this URL.

37.3. Validate installation

Restart the Eclipse IDE after the installation. Ensure that your plug-in is available in your Eclipse installation and can be used.



38
Patching existing Eclipse plug-ins with feature patches

38.1. Feature patch projects

If you want to modify standard Eclipse plug-ins, you can use *Feature Patch Projects*. A *Feature Patch Project* can contain plug-ins which replace existing plug-ins. This would allow you to create an update site for this feature and users can use your version of the modified code.

Each feature can only be patched once.

38.2. Feature patch projects

A feature patch project can get created via *File* → *New* → *Other...* → *Plug-in Development* → *Feature Patch*.

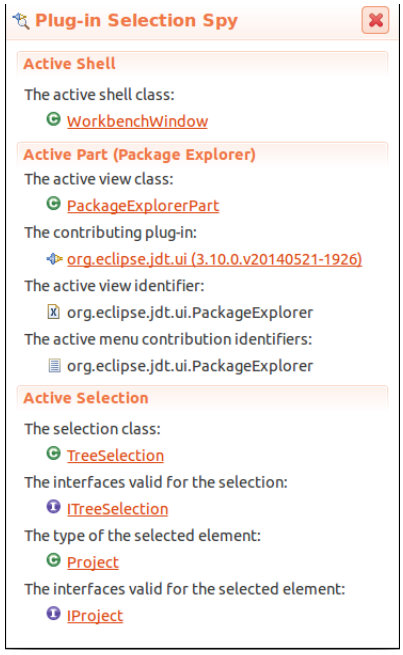
In this dialog select the feature which you want to patch. Add every modified plug-in to your new *Feature Patch Project* and export it as a p2 update site.

Part VI. Tools for accessing the Eclipse source code

39
Eclipse spy functionality for finding things

39.1. Plug-in Spy for UI parts

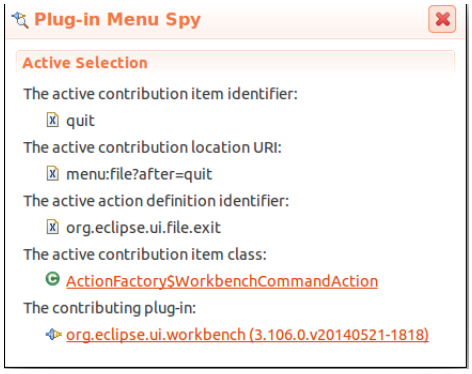
You can activate the *Plug-in Spy* by pressing **Alt**+**Shift**+**F1** in the Eclipse IDE. It gives you information about the currently selected user interface component. This way you can get immediate access to the plug-in which is currently running.



Click on any of the linked elements to obtain more information about that element. For example, if you click on the *contributing plug-in* the tool opens the manifest editor for this plug-in.

39.2. Menu spy

Press **Alt**+**Shift**+**F2** and select a menu entry or click a toolbar entry to see information about this element.



39.3. SWT Spy

SWT Spy for Eclipse is a tool that prints out information about the widget under the cursor. Currently, this includes style, layout and parent information. See *SWT Development Tools homepage [http://www.eclipse.org/swt/tools.php]* for more information.

40
Using the e4 spies

40.1. The e4 tools project

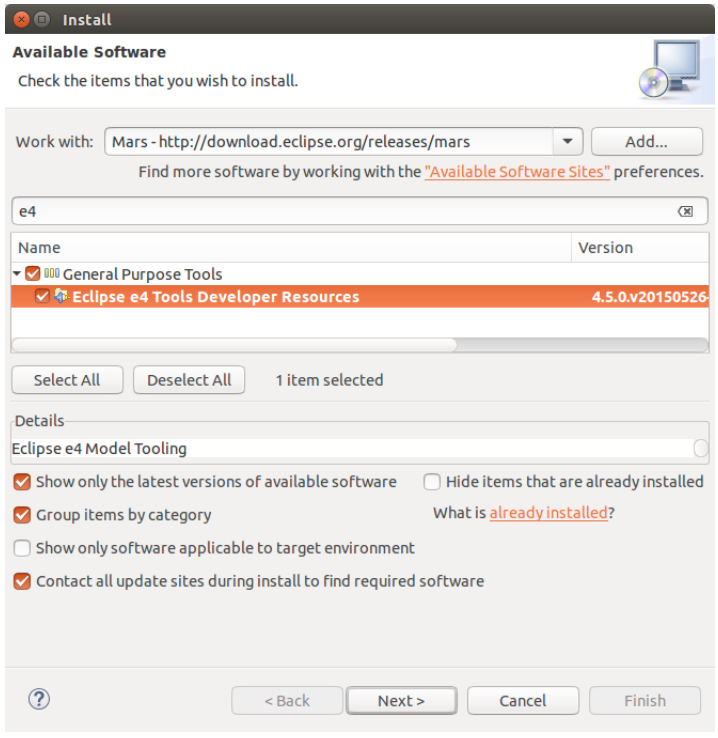
The e4 tools project provides more tools to analysis an Eclipse based application, including the Eclipse IDE.

40.2. How to install the e4 tools

40.2.1. Install the e4 tools

The e4 tools provide the tools to develop Eclipse 4 RCP applications. These tools provide wizards to create Eclipse application artifacts and an application model editor.

These tools can be installed into the Eclipse SDK via the Eclipse default update site.



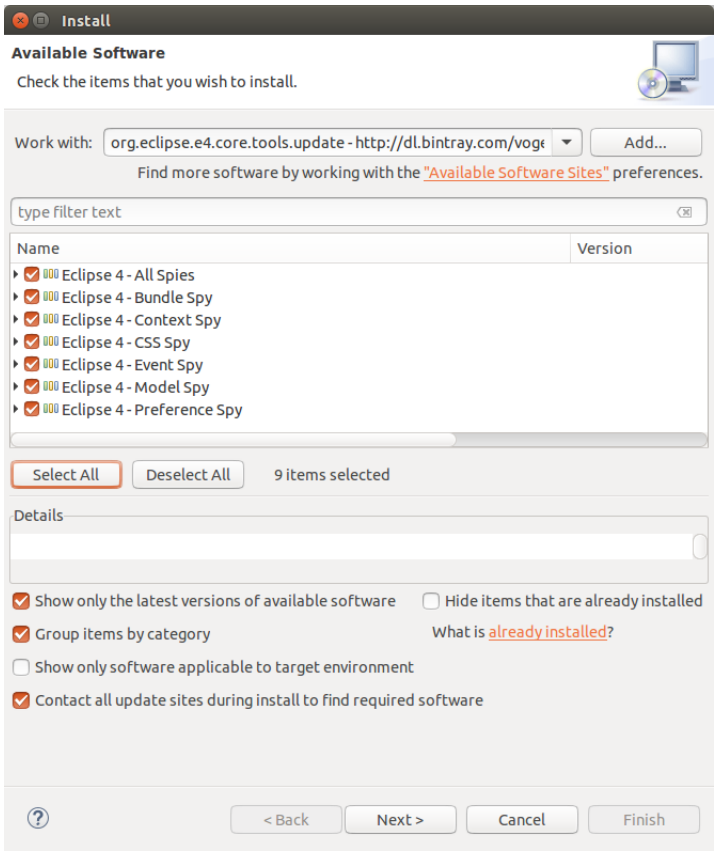
Install the tools and restart your Eclipse IDE after the installation.

40.2.2. Install the e4 spies from the vogella GmbH

The e4 tools update site from above contains the core tools to develop e4 applications. More tools to analyze Eclipse RCP applications can also be installed. The vogella GmbH company provides a recent version of these tools for the Eclipse 4.5 release under the following URL:

```
http://dl.bintray.com/vogellacompany/e4tools
```

You can install the e4 tools via *Help* → *Install New Software...* by entering the URL. The e4 tools installation from the vogella update site is not signed.



40.2.3. Install the e4 spies from Eclipse.org

The vogella GmbH update site was created to provide a stable link for users of the e4 tools. The same code base is used by Eclipse.org to create an official update site for the *Eclipse e4 tooling*. Unfortunately the link for this update site changes from time to time but it can be found on the following website: *Eclipse.org e4tools site [http://download.eclipse.org/e4/downloads/]*.

If you click on a *Build Name* link, you also find the URL for the update site. The following screenshots demonstrate this for a particular build of the e4 tools.

Note

This website might change over time.

Tip

The content of this update site might change. The *e4 tools info page [http://e4tools.vogella.com/]* contains the latest information about the e4 tools update site. Especially for Eclipse 4.5 you should check this side, as the process of building the tools is planned to be changed in Eclipse 4.5.

40.3. Analyzing the application model with the model spy

The application model of an Eclipse application is available at runtime. The application can access the model and change it via a defined API.

To analyzing this application model and for testing model changes, you can use a test tool from the e4 tools project which allows modifying the application model interactively. This tool is called *Model Spy*

(used to be called: Live model editor) and can be integrated into your RCP application. Most changes are directly applied, e.g., if you change the orientation of a part sash container, your user interface is updated automatically.

In the model spy, you can select a part in the application model, right click on it and select *Show Control* to get the part highlighted.

40.4. Model spy and the Eclipse IDE

If installed you can also use the model spy to see the application model of the running Eclipse IDE itself. You can open it via the **Alt**+**Shift**+**F9** shortcut.

> **Warning**
>
> If you modify the Eclipse IDE model, you should be careful as this might put the running Eclipse IDE into a bad state. To fix such issues, start the Eclipse IDE from the command line with the `-clearPersistedState` parameter.

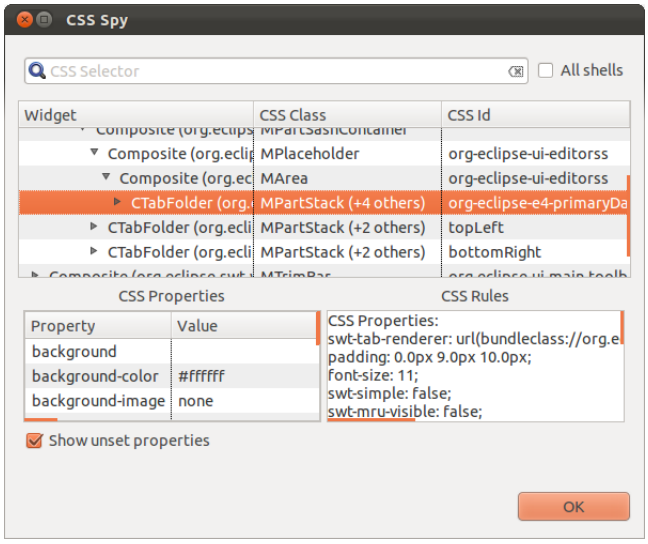
40.5. CSS Tools

40.5.1. CSS Spy

The *CSS spy* tool helps to see the possible styling options for selected elements. *CSS spy* is part of the *e4 tooling* project and can be installed from its update site. See the Eclipse 4 installation description for details.

You can open the *CSS spy* via the **Shift**+**Alt**+**F5** shortcut.

CSS spy allows you to see the available properties and the current style settings.



40.5.2. CSS Scratchpad

The *CSS Scratchpad* allows to change CSS rules interactively. is also part of the *e4 tooling* project.

You open it via the **Ctrl**+**Shift**+**Alt**+**F6** shortcut.

If you click the *Apply* button, the entered CSS is applied at runtime to your application.

41
Using the search functionality

41.1. Filtering by the Java tools

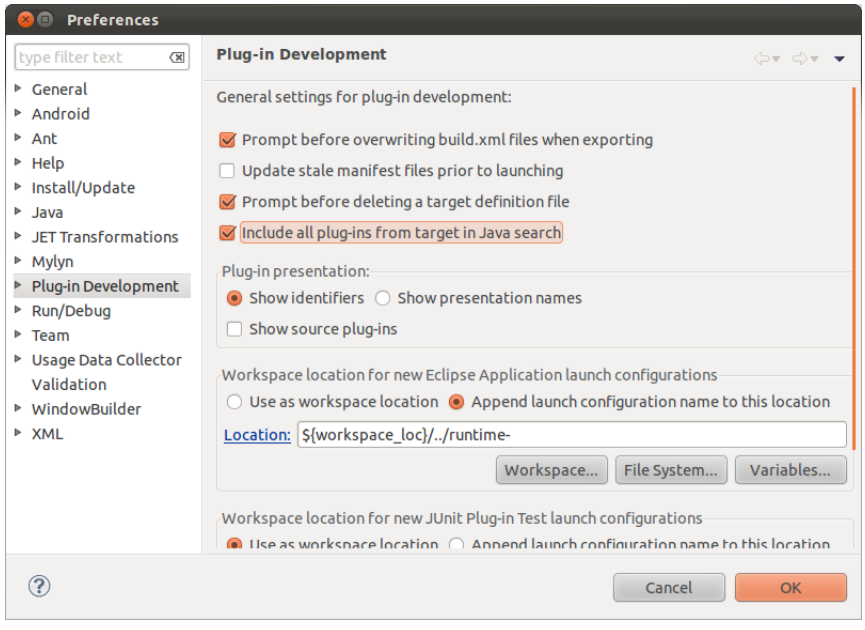
The Java Development Tools (JDT) of the Eclipse IDE limit the scope of search related activities. By default, JDT considers elements from opened projects including their dependencies as well as elements from the standard Java library.

For example, the **Ctrl**+**Shift**+**T** (Open Type) shortcut will not find the `ISources` interface if it is not referred to in the current workspace. As plug-in developer you want to have access to all classes in your current *target platform*.

The *target platform* is the set of plug-ins against you develop. By default, the plug-ins from the Eclipse IDE installation are used as target platform.

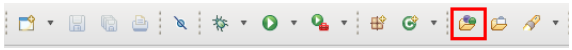
41.2. Include all plug-ins in Java search

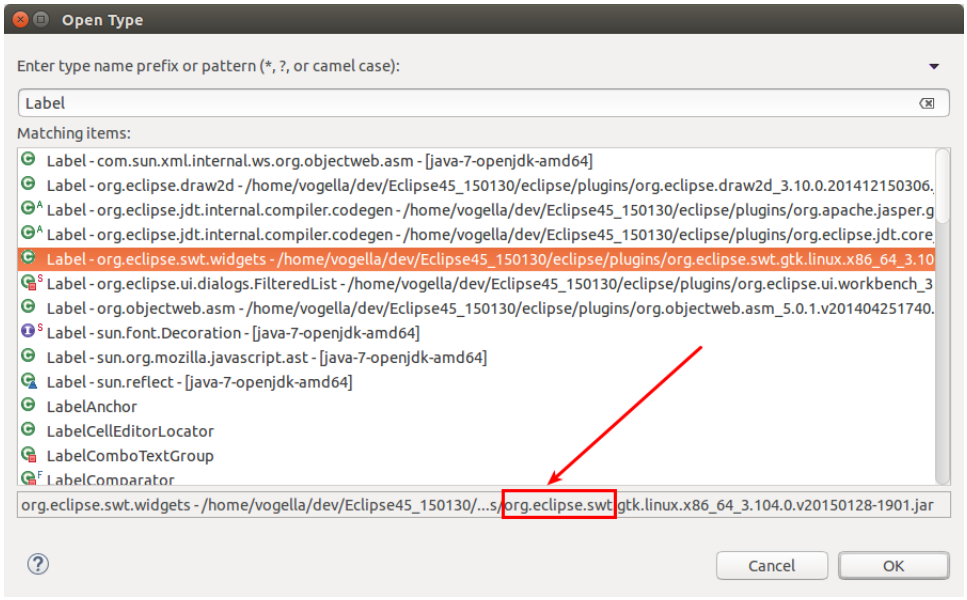
You can include all classes from the current target platform to be relevant for the Eclipse Java tools via the following setting: *Window → Preferences → Plug-in Development → Include all plug-ins from target in Java search.*



41.3. Find the plug-in for a certain class

You frequently have to find the plug-in for a given class. The Eclipse IDE makes it easy to find the plug-in for a class. After enabling the *Include all plug-ins from target into Java Search* setting in the Eclipse IDE preferences you can use the *Open Type* dialog (**Ctrl**+**Shift**+**T**) to find the plug-in for a class. The JAR file is shown in this dialog and the prefix of the JAR file is typically the plug-in which contains this class.

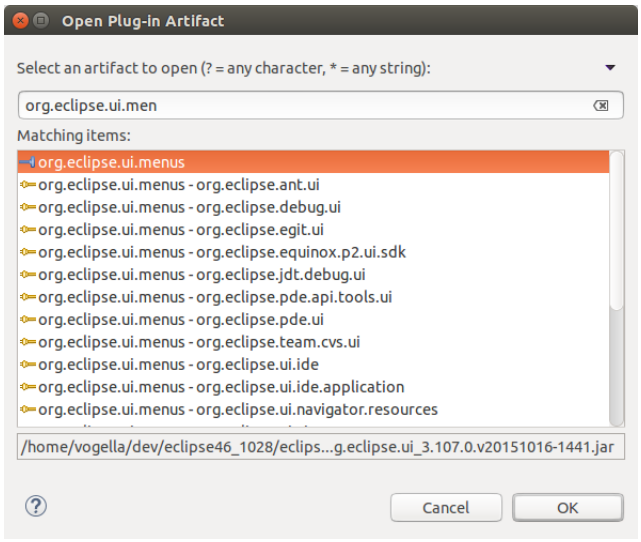




41.4. Finding classes and plug-ins

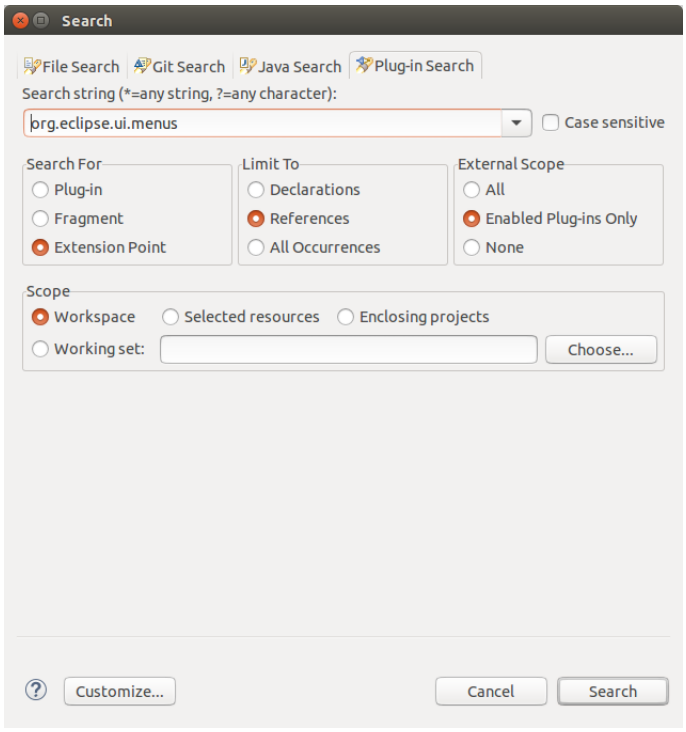
41.4.1. Open Plug-in artifact

Use the **Ctrl**+**Shift**+**A** shortcut to search for an extension point. Definition and usage of an extension point have different icons. For example, to see which plug-in defines the `org.eclipse.menu.ui` extension point, enter it in the dialog and select the entry with the blue icon.



41.4.2. Plug-in search

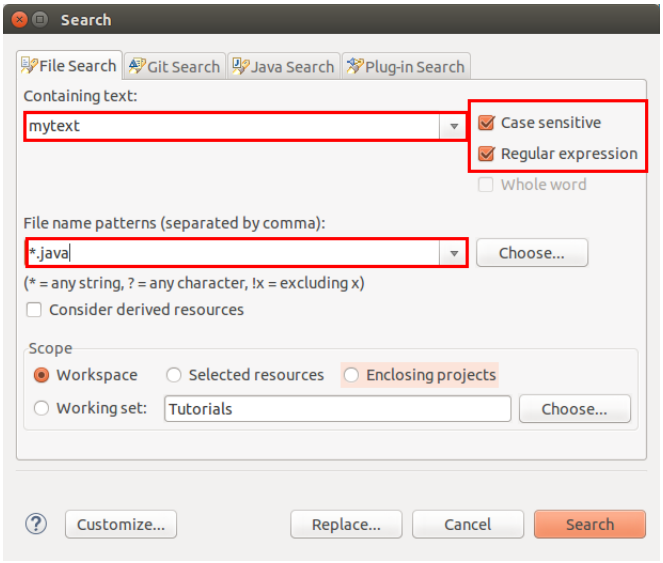
Plug-in Search allows you to perform a detailed search for extension points. Select the *Search* → *Search* → *Plug-in Search* menu. You can specify what your are searching for as demonstrated in the following screenshot.



41.4.3. Plain text search

In case you have imported the source code of an Eclipse project into your workspace you can also use the plain text search. Select the *Search* → *Search* menu entry and switch to the *File Search* tab.

As indicated in the following screenshot you can search for a text, use regular expressions and restrict which files to search by specifying a file name pattern. This is a very flexible way to search and allows you to find almost everything.



41.5. Example: tracing for key bindings

The tracing functionality of Eclipse allows to you trace which command is associated with a certain key binding. The following listing contains the trace options to enable that.

```
# turn on debugging for the org.eclipse.core.resources plugin.
org.eclipse.ui/debug=true
org.eclipse.ui/trace/keyBindings
org.eclipse.ui/trace/keyBindings.verbose
```

42
Tracing

42.1. What is tracing?

Eclipse provides a tracing facility which can be activated on demand. If turned on, the Eclipse IDE writes additional information to a specified log file or the console.

42.2. Turning on tracing via an options file

To turn tracing on, you need to create a trace-options file that contains key-value pairs for the tracing options which should be turned on.

By default, this file is called .options and the Eclipse runtime looks for such a file in the Eclipse install directory. The text should contain one key=value pair per line. To turn on the trace options in the preceding two examples, you need an options file that looks like this:

```
# turn on debugging for the org.eclipse.core plugin.
org.eclipse.osgi/debug=true

# turn on loading time for the plug-ins
org.eclipse.osgi/debug/bundleTime=true
```

You can use the corresponding preference dialog to find the values you want to enable. See Section 42.3, “Turning on tracing at runtime”.

Finally, you need to enable the tracing mechanism by starting Eclipse with the `-debug` command line argument. You can optionally specify the location of the options file as a URL or a file-system path after the -debug argument.

```
# start Eclipse with the test workspace
# piping the output into a file
./eclipse -debug -data ~/test > trace.txt
```

For example the *Starting application* describes when OSGi is done with its initialization and the *Application Started* tells you when the application has been started. Afterwards you can extract the information which interest you the most. Here are a few commands which extract the starting time of each bundle and sort the bundles by this time.

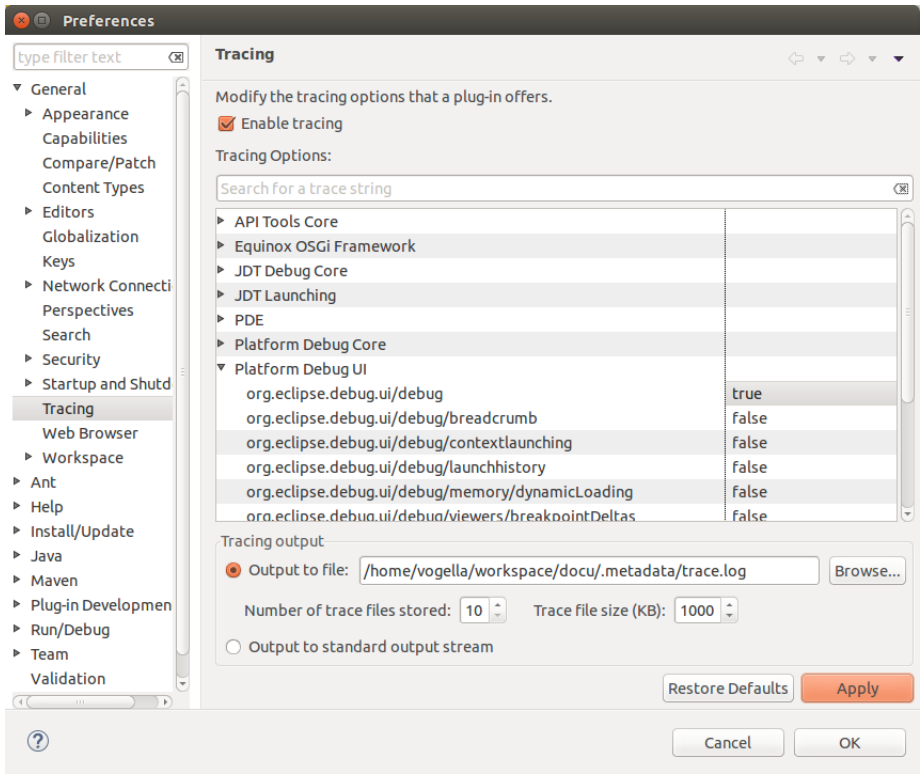
```
# extract the lines starting with "End starting"
grep "End starting" trace.txt  >trace2.txt
# extract plug-in name and startup time and sort by time
awk '{print $5 " " $3}' trace2.txt | sort -nr > tracefinal.txt
```

The following is another example for an .option file in which you trace the resources.

```
# turn on debugging for the org.eclipse.core.resources plugin.
org.eclipse.core.resources/debug=true

# monitor builders and gather time statistics etc.
org.eclipse.core.resources/perf/builders=10000

# monitor resource change listeners and gather time statistics etc.
org.eclipse.core.resources/perf/listeners=500

# monitor workspace snapshot and gather time statistics etc.
org.eclipse.core.resources/perf/snapshot=1000

# monitor workspace snapshot and gather time statistics etc.
org.eclipse.core.resources/perf/save.participants=500

# debug build failure cases such as failure to retrieve deltas.
org.eclipse.core.resources/build/failure=true

# reports the cause of autobuild interruption
org.eclipse.core.resources/build/interrupt=true

# reports the start and end of all builder invocations
org.eclipse.core.resources/build/invoking=true

# reports the start and end of build delta calculations
org.eclipse.core.resources/build/delta=true

# for incremental builds, displays which builder is being run
# and because of changes in which project.
org.eclipse.core.resources/build/needbuild=true

# prints a stack trace every time an operation finishes that requires a
build
org.eclipse.core.resources/build/needbuildstack=true

# prints a stack trace every time a build API method is called
org.eclipse.core.resources/build/stacktrace=false

# report debug of workspace auto-refresh
org.eclipse.core.resources/refresh=true
```

42.3. Turning on tracing at runtime

It is also possible to turn on some tracing options at runtime via the Eclipse IDE preferences as depicted in the following screenshot.


Preferences
type filter text
General
Appearance
Capabilities
Compare/Patch
Content Types
Editors
Globalization
Keys
Network Connecti
Perspectives
Search
Security
Startup and Shutd
Tracing
Web Browser
Workspace
Ant
Help
Install/Update
Java
Maven
Plug-in Developmen
Run/Debug
Team
Validation
Tracing
Modify the tracing options that a plug-in offers.
Enable tracing
Tracing Options:
Search for a trace string
API Tools Core
Equinox OSGi Framework
JDT Debug Core
JDT Launching
PDE
Platform Debug Core
Platform Debug UI
org.eclipse.debug.ui/debug
true
org.eclipse.debug.ui/debug/breadcrumb
false
org.eclipse.debug.ui/debug/contextlaunching
false
org.eclipse.debug.ui/debug/launchhistory
false
org.eclipse.debug.ui/debug/memory/dynamicLoading
false
Tracing output
Output to file:
/home/vogella/workspace/docu/.metadata/trace.log
Browse...
Number of trace files stored:
10
Trace file size (KB):
1000
Output to standard output stream
Restore Defaults
Apply
Cancel
OK

43
Updating the copyright header of a source file

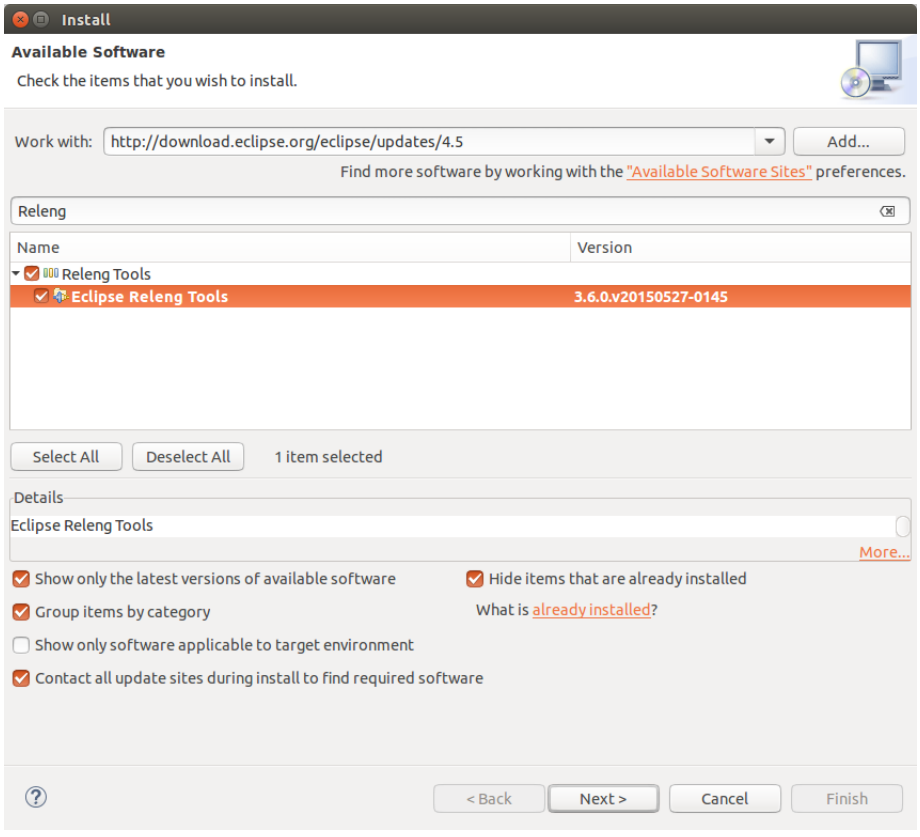
43.1. Why updating the copyright header of a changed file?

Most Eclipse projects insists that you update the last changed date in the copyright header if you change the file. You can do this manually but the Eclipse releng project also provides tooling for that.

43.2. Using the Eclipse releng tooling for updating the copyright header

You need to install the copyright header tool from the following update site: http://download.eclipse.org/eclipse/updates/4.5

Adjust the number "4.5" to the number matching your eclipse version.



After the installation you can select the entry from the context menu of a project or source folder.

Part VII. Building your custom IDE distribution

44
Creating a custom Eclipse IDE build

44.1. Building the Eclipse IDE

Eclipse uses a Maven based build system for automated build using the Maven Tycho plug-in. Using this build system you can create your custom build of the Eclipse IDE. If you building the Eclipse IDE from the current master branch you get an IDE based on the latest developments of the Eclipse development team.

This effort is part of the Common Build Infrastructure (CBI) project which provide tools to simplify and standardize the Eclipse build process.

The results of this build are archive files for the different platforms, which include everything to run an Eclipse IDE.

44.2. Requirements

As the build continuously changes, the results and requirement of the build might be slightly changed at the time you read this. See Section 45.1, “Reporting issues or asking questions about the build” for finding additional information.

Depending on your network connection and your machine power the build of the Eclipse IDE takes around 2 hours on a Core i5 machine with SSD, so of course time is required for this. Also, approximately 25 Gigs of free space and 4 Gigs of RAM are required on the hardware-side. Building of an Eclipse IDE is possible on Windows, OS X and Linux based distributions. This tutorial is tested with Linux.

On the software-side the following software is required:

- Git
- Maven Version 3.3.1
- Oracle 1.8 JDK or higher

44.3. Cloning the SDK repository

You download the newest version of the source code by cloning the following repository and it's submodules via Git.

```
git clone -b master
   --recursive git://git.eclipse.org/gitroot/platform/eclipse.platform.releng.aggregator.git
```

44.4. Building the Eclipse IDE

You start the build with the following command.

```
mvn clean verify
```

If you receive `java.lang.OutOfMemoryError` error during the Maven build, you should increase the memory which is available for the build.

```
export MAVEN_OPTS="-Xmx2048m"
```

If the build is successful, the Eclipse SDK is packaged as archive files for all supported platforms. These packages can be found in the Git repository in the following folder:

```
eclipse.platform.releng.tychoeclipsebuilder/sdk/target/products/*
```

44.5. Cleanup before the next build

To ensure that the repository is clean and up to date we execute the following Git commands.

```
git submodule foreach git clean -f -d -x
git submodule foreach git reset --hard HEAD
git clean -f -d -x
git reset --hard HEAD
```

The build requires that the version numbers of each single Maven artifacts and it's Eclipse plug-ins are in sync. But sometimes (this is rare these days) developers forget to update the Maven version numbers. In this case the build complains about a version mismatch. Run the following command to correct existing inconsistencies.

```
mvn -Dtycho.mode=maven org.eclipse.tycho:tycho-versions-plugin:update-pom
```

45
Additional information about building the Eclipse platform

45.1. Reporting issues or asking questions about the build

A good place to post questions about the CBI build is the CBI mailing list. The webinterface for this mailing list can be found under the following URL *CBI mailing list [https://dev.eclipse.org/mailman/listinfo/cbi-dev]*.

45.2. Eclipse platform Hudson builds

The Eclipse platform has a Hudson build instance which can be found under the following URL: *Eclipse platform Hudson [https://hudson.eclipse.org/platform/]*. This build instance is currently not used for the main build, only to validate Gerrit contributions.

See *Bug report for migrating the platform build to Hudson [https://bugs.eclipse.org/bugs/show_bug.cgi?id=420101]*.

45.3. Release and milestone builds

To have a reproducible and stable build, it is possible to switch to a specific version instead of using the newest source code. For this the platform aggregator repository has tags for every release build.

To build for example the R4_3 release, you have to checkout this specific tag out and start the build like the following:

```
git checkout tags/R4_3
git submodule update
#run the build
mvn clean verify
```

Warning

The CBI build still depends on external resources like p2 update sites. From time to time this resources change or will become unavailable. The R4_3_1 tag for example depends on a removed update site, so you have to add the new URL via a parameter to the build:

```
git checkout tags/R4_3_1
git submodule update

# run the build
# command must be issued in one line
mvn clean verify
 -Dlicense-repo.url=http://download.eclipse.org/cbi/updates/license/
```

Tip

To check which properties can be overridden check the properties fields in the eclipse-platform-parent/pom.xml file.

45.4. Changing build ID

A normal CBI Build shows, "Build id: @build@" in the "About Eclipse SDK" window. To define a custom build id use the `buildId` parameter and the `update-branding-plugins` profile.

```
mvn clean verify -DbuildId=foobar -Pupdate-branding-plugins
```

45.5. Build single parts of the aggregator

It is also possible to build just single parts of the aggregator repository, by passing the `build-individual-bundles` profile. The following example shows how to build just the rt.equinox.framework.

```
cd rt.equinox.framework \
mvn -Pbuild-individual-bundles clean verify
```

Note

Unfortunately while there are numerous bundles for which the build succeed with this method, there are a couple bundles / features that cannot be built using this method.

45.6. Building natives (SWT binary files)

The CBI build does not build natives by default (SWT binary files). Instead, the CBI build simply copies pre-built native files from the swt.binaries repository.

For Windows/Mac/Linux it is possible to force the CBI build to build these natives by passing the -Dnative pointing to the build. For example the *`-Dnative=gtk.linux.x86_64`* parameter in the case of Linux. List of options available at *Building natives [https://wiki.eclipse.org/Platform-releng/Platform_Build#Building_natives]*.

45.7. Fedora Eclipse CBI build

The Fedora project has removed some non Eclipse IDE related plug-ins to speed up the build process. See the *Fedora build script [http://pkgs.fedoraproject.org/cgit/eclipse.git/tree/eclipse.spec#n366]*.

Part VIII. Eclipse foundation staff and Eclipse project lead interviews

46
Mike Milinkovich about the vision of Eclipse

46.1. Who are you and what is your role in the Eclipse organization?

I am the Executive Director of the Eclipse Foundation, which means that I have overall responsibility for the operations of the Eclipse Foundation and the Eclipse Management Organization (EMO).

What that means on a day-to-day basis is that I am working on making Eclipse a great place to host open source projects, while at the same time helping member companies build successful businesses with our technology. I sometimes call myself Eclipse's chief cheerleader, as I spend a lot of my time promoting Eclipse around the world.

I have been in this role for ten years now, and have flown over 1.2 million miles helping raise awareness of Eclipse and our outstanding community.

46.2. Where do you see the cornerstones of the Eclipse OS project?

The definition of what it means to be an Eclipse project has evolved over the past ten years. Originally an Eclipse project was an extensible tool plug-in that was written in Java and used the OSGi-based Eclipse plug-in model. When the Rich Client Platform (RCP) came along in 2004, the definition evolved to desktop runtimes. Then soon after with Equinox, Gemini and Virgo, the definition evolved to include server-side runtimes as well.

Today, the Eclipse Foundation will accept projects written in any programming language, on any platform.

The distinguishing feature of the Eclipse open source community has become our development process, and our intellectual property (IP) management process.

So we have evolved from a definition for Eclipse projects that was entirely technology-based to a definition which is community-based. We are attracting projects which wish to share in Eclipse's

community of practice on how to build open source projects under a disciplined and predictable development process, and which have their IP thoroughly reviewed and vetted by a professional team.

46.3. What is your vision for the future of the Eclipse OS project?

There is so much going on in the Eclipse community these days, it's hard to narrow this down! But here are a couple of key areas where I think we're going to see a lot of activity over the next couple of years.

- Tooling, especially tooling for Java developers is going to see a resurgence of investment and improvements. The team has been entirely focused on implementing Java 8 support over the past year or so. Now they have some time to focus on UI and performance improvements. The work that we have done as a community to lower the barriers to contribution are also starting to pay off, and we are starting to see more external contributions to the tooling platform as well. The recently proposed Flux and Che projects are also creating an incredibly cool community of web-based development tooling projects for Java developers as well. We see Eclipse at the forefront of the movement to build seamless developer experiences across the desktop and browser-based tooling platforms.

- Modeling continues to be a leading part of the Eclipse community as well. In addition to the traditional modeling project community, we see the Polarsys working group as a very important extension. Polarsys uses the Eclipse modeling projects to create toolchains for embedded development of critical systems, along with very long-term support for those tools.

- The Internet of Things (IoF)is a rapidly growing area, and the Eclipse Foundation is the leading community focused on creating a free and open infrastructure for the IoT. With fifteen projects and counting, Eclipse IoT is providing protocols, frameworks, runtimes and tools for developers working on IoT solutions.

- In general, we see our collaborative working groups as a key growth area for Eclipse. In areas as diverse as automotive and geospatial technologies, Eclipse's working groups are bringing new members, projects and technologies to our community.

47
Wayne Beaton and Ian Skerrett about the Eclipse community work

47.1. Who are you and what is your role in the Eclipse organization?

I'm Wayne Beaton. My official title is Director of Open Source Projects. I see that role as primarily being one of helping open source developers be successful with their projects. In practical terms, I maintain the process and try to ensure that projects are operating according to the open source rules of engagement; in particular, that projects establish a level playing field that invites participation. In the past, I did evangelism. I'm expanding my role back into that space.

My name is Ian Skerrett. I help coordinate the marketing activities of the Eclipse Foundation. This means I try to help work promote what Eclipse projects are doing and also what Eclipse Foundation members ’are doing with Eclipse.’

47.2. Why should someone consider contributing to Eclipse?

Wayne: Adventure! Excitement!

I assume that you mean Eclipse in the larger sense; that of a community of open source projects managed by the Eclipse Foundation. We have a lot of interesting technology being developed here, so from a pure technical interest point of view, we have a lot of offer developers.

I'll admit that the ramp up process for contributors is still more difficult that we'd like, but we have a good support network here. There's a certain consistency to finding and building code. Once you are in, there's people in the community who will step up to help. Of course some of our project teams are

better at engaging with the community than others, but it's something that we at the Eclipse Foundation work constantly at improving.

47.3. How does the Eclipse foundation helps the Eclipse projects?

Wayne: We play a bit of a match maker role. The Eclipse Foundation staff tends to have a broad view of the community, so we notice when individuals and organizations have similar goals. We help these people meet each other.

We also continually strive for improvements in our processes to lower barriers and make it easier for contributors to be successful.

Ultimately, it is the open source project teams themselves who have the biggest impact on whether the project remains active; we can make the introductions, and reduce the process barriers, but we depend on the project teams to accept contributions and engage contributors.

47.4. Which tools do you provide which are most valuable for developers?

Wayne: We provide a lot of tools and services for committers. Issue tracking, code review, and build support are probably some key services.

The most valuable, however, is communication channels. We provide transparent and open channels for open source project teams to communicate among themselves and with their communities, foster growth, and diversity. This includes teaching project teams to operate in a transparent and open manner.

A lot of our recent efforts, especially around the new project management infrastructure (the so-called "PMI") is about making it generally easy for contributors to orient themselves and ultimately be successful. We're making good progress, I think, in terms of making it easier for contributors to find code, run builds, and make contributions.

I include our adoption of Gerrit in this as well; Gerrit makes it easy to interact with the open source project teams and contribute.

47.5. What improvements can contributors and committers expect?

Wayne: We continually work to improve the tools and services we provide. So committers will see regular improvements in things like Hudson and Gerrit support.

Social coding is something that we feel is important to the community, so we've rolled out the ability for Eclipse open source projects to use GitHub as the canonical host for their source code. We currently don't allow projects to use GitHub issues, but it's something that we'd like to make possible.

We're in the process of rolling out a new metrics dashboard for Eclipse projects, and investigating the official deployment of Sonar.

We're pushing hard to move our interaction points with committers out of the old developer portal and into the PMI. Our hope is that user interface consistency will make it easier for committers to work through the process bits required of them.

47.6. What role does marketing play in the Eclipse foundation?

Ian: One of the great things about open source and the Eclipse community is that the best marketing comes from our community. Developers talk to other developers about what they like (and dislike) about Eclipse. They help each other to be successful with Eclipse and this is the best marketing you can get.

The great thing about community marketing is that it is authentic and relevant to what developers care about. The most successful projects at Eclipse enable their community to do their marketing.

47.7. If someone wants to do a community event, how should he get into contact with the foundation?

Ian: They should email events@eclipse.org.

48
Denis Roy about the Eclipse infrastructure

48.1. Who are you and what is your role in the Eclipse organization:?

My name is Denis Roy and my title is "IT Director" but I'm really a network and systems administrator for the Eclipse Foundation.

I lead the technical aspects of the servers, software and network infrastructure that are used by the Eclipse community. Most recognize me by the email address, webmaster@eclipse.org.

48.2. Why should someone be interested in contributing to Eclipse?

Eclipse is many things: a flexible platform on which software can be built; a large assortment of plugins and tools, and a great community. It's also a development environment used to build software that can be found in products we use on a daily basis, such as mobile devices, video games, popular websites and even in our cars.

48.3. Which tools are the most valuable for committers?

I think the combination of Git, Gerrit and Hudson provide optimized source control management, code review and a quasi-instant CI feedback loop that allow committers to collaborate on new features and bugs effectively.

48.4. Which tools are most valuable for contributors?

There are three that come to mind: Bugzilla, Forums and Gerrit. The Forums provide a simple interface for the typical first interaction between a potential contributor and the project team, while Bugzilla is used by users, contributors and committers alike to provide support and enable interactive discussions. Gerrit allows contributors to suggest patches and get feedback which can lead to more active participation. By creating and commenting on bugs, talking to users on the forums and submitting patches individuals are stepping up and providing valuable information, feature requests and other feedback to the committers.

49
The Java development tools (JDT) project

49.1. Can you describe the target and scope of the JDT project?

The JDT project provides the plug-ins that implement a Java IDE supporting the development of any Java application, including Eclipse plug-ins. It adds a Java compiler, Java project nature, and Java perspective to the Eclipse Workbench as well as a number of views, editors, wizards, builders, and code merging and refactoring tools.

The JDT project is divided into the JDT Core, JDT Debug, and JDT UI sub-projects.

49.2. Who are you and how are you involved in the Eclipse community?

I'm Dani (Daniel Megert) and I'm one of the initial Eclipse committers. Currently I'm leading the Platform and the JDT projects. I'm a member of the *Eclipse Project PMC [https://www.eclipse.org/eclipse/team-leaders.php]* and represent the Eclipse project in the *Planning Council [https://wiki.eclipse.org/Planning_Council]*.

49.3. Can you describe the project team and culture?

As of October 19, 2015, the 9 active JDT committers are distributed over 4 different countries and 4 companies, with IBM being the largest contributor. We have an open and performance-oriented culture that is based on meritocracy.

An important part of our openness is that we keep all our discussions and decisions in Bugzilla, so that others can participate.

49.4. Where can a contributor find information how to contribute?

The JDT project pages list the related Git repositories under *Developer resources*.

- https://projects.eclipse.org/ provides all relevant information, like plans, releases, committers, etc
- https://projects.eclipse.org/projects/eclipse.jdt
- https://projects.eclipse.org/projects/eclipse.jdt.core

- https://projects.eclipse.org/projects/eclipse.jdt.debug
- https://projects.eclipse.org/projects/eclipse.jdt.ui

Each sub-project has its own wiki page for this *JDT Core [https://wiki.eclipse.org/JDT_Core_Committer_FAQ]*, *https://wiki.eclipse.org/JDT_UI/How_to_Contribute [JDT UI]* and *https://wiki.eclipse.org/Debug/Developers [JDT Debug]*. Common to all is the *wiki [https://wiki.eclipse.org/Platform-releng/Git_Workflows]* about how we use Git in the Platform, JDT and PDE.

The most important thing for a contributor to keep in mind is that we expect production-ready contributions, and for this it is best to exactly follow the contributor guidelines (see above). Here are some specific points:

1. Consider legal issues
   - All code, icons, documentation etc. that has not been authored by the contributor must be indicated as such
   - Update copyright notice and date in all changed files, including properties files.
2. Adhere to JDT's naming conventions and honor the style conventions, like compact assignments
3. Make sure your code is formatted
   - this should happen automatically if you use save actions
   - however: do not format the whole file - only format the code you add or touch
4. Don't submit a patch that adds any warning or error
5. Write JUnit test cases where it makes sense and contribute them as well
6. Reuse existing UI metaphors and reuse/generalize code rather than copy/paste or reinvent the wheel

50
The Eclipse Git project

50.1. Can you describe the target and scope of the EGit project?

EGit is an Eclipse Team provider for the Git version control system. The EGit project is implementing Eclipse tooling on top of the JGit Java implementation of Git.

JGit is an *EDL [http://www.eclipse.org/org/documents/edl-v10.php]* (new-style BSD) licensed, lightweight, pure Java library implementing the Git version control system. JGit has very few dependencies, making it suitable for embedding in any Java application, whether or not the application is taking advantage of other Eclipse or OSGi technologies. It's used in many applications.

50.2. Who are you and how are you involved in the Eclipse community?

My name is Matthias Sohn and I am co-leading the Eclipse EGit and JGit projects together with Shawn Pearce from Google. I also contribute to Orion and Gerrit Code Review.

I'm working at SAP and leading its Git / Gerrit team.

50.3. Can you describe the project team and culture?

We have team members from a good variety of Eclipse member companies and also individual committers who aren't affiliated to a member company. We are spread across the globe and keep in touch using code review, Bugzilla and mailing list. Many of us also use Gerrit at work and also contribute to it.

JGit has 15 committers and EGit has 23 committers and we receive many "drive by" patches, per release (every 3 months) we have contributions from 15-30 developers per project.

JGit is used in many applications, contributions are mainly driven by the needs of these applications. Most contributions are driven by the needs of the developers using the Git tooling in Eclipse. Code review is at the center of our interactions around code.

We practice code review for all changes, except release engineering changes done when creating a release. We release every 3 months and participate in the Eclipse simultaneous releases. The team members frequently give talks and tutorials at EclipseCons and Demo camps, this is also the chance to meet committers, contributors and users face to face and often ideas for new features stem from such events.

50.4. Where can a contributor find information how to contribute?

The Git repository for Eclipse Git can be found on the *EGit project page [https://projects.eclipse.org/projects/technology.egit/developer]*. The contribution process is described by *EGit contributor guide [https://wiki.eclipse.org/EGit/Contributor_Guide]*.

The main things for a prospective EGit contributor to consider are:

1. Ensure the license of the contributed code permits this contribution
2. Make separate commits for logically separate changes. Describe the technical detail of the change(s). If your description starts to get too long, that's a sign that you probably need to split up your commit to finer grained pieces.
3. Write tests
4. Send your changes using Gerrit as it facilitates the work of reviewers.

51
The m2e project for Eclipse Maven integration

51.1. Can you describe the target and scope of the m2e project?

The goal of the m2e project is to provide tight integration between the Eclipse IDE and Maven. This includes user-facing features like pom.xml editor and support for standard Maven plugins. There is also a rich extension API that third party developers can use to integrate their Maven plugins and tools into the Eclipse IDE.

51.2. Who are you and how are you involved in the Eclipse community?

My name is Igor Fedorenko. I am originally from Omsk, Russia but currently live in Toronto, Canada. For the last 7 or so years I mainly work on Maven related development tools. I started to use Eclipse around 2.0 M1 and contribute patches not too long after that. I currently work at takari.io. Our goal is to drive innovation in the area of continues software release and delivery and software development tools and processes in general.

I am involved in m2e and related project both hosted at eclipse and outside. I also participate in Tycho development, although not as actively and worked on the Eclipse CBI project and platform build in the past.

Outside of Eclipse, I work on Maven core and Maven incremental build solutions.

51.3. Can you describe the project team and culture?

Jason van Zyl and myself are the project co-leads. There are currently two active committers.

In m2e 1.5, our most recent release, there were contribution from 5 non-committers. One of these 5 contributors provided three separate patches.

m2e does not have any development plans. All work happens in ad-hoc fashion. I am a heavy m2e user myself, so almost all m2e changes I make are driven by my own needs, either bugs I run into or features I need. Although we appreciate bug reports, we encourage users to provide patches. Bug reports that have not had any meaningful activity for 12+ months are closed automatically.

51.4. Where can a potential contributor find information about potential contributions?

The *M2E development environment Wiki [http://wiki.eclipse.org/M2E_Development_Environment]* explains how to setup the m2e development environment, build m2e on the command line and how to run the integration tests and prepare and submit patches.

The m2e-users and m2e-dev mailing lists is where we discuss everything related to the m2e project.

52
The C and C++ IDE (CDT) project

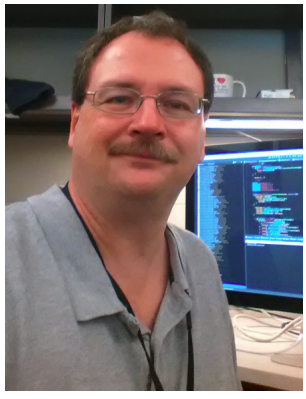
52.1. Can you describe the target and scope of the CDT project?

The CDT Project provides an industrial-strength C and C++ Integrated Development Environment based on the Eclipse platform.

With conservative statistics reaching a million direct downloads yearly, the CDT is the defacto standard C/C++ IDE, especially in the embedded space. It is also a platform for most commercial adopters to provide value-added tooling for C/C++ developers. Furthermore, it serves as a foundation for other Eclipse projects such as the Linux Tools project and the Parallel Tools project.

The CDT provides features such as: project and build, full-featured editor, source navigation, refactoring, static code analysis, debugging, unit testing.

52.2. Who are you and how are you involved in the Eclipse community?



Doug Schaefer has been leading the CDT since 2005 and has been involved in it since its first beginnings in 2002. He is part of various Eclipse Councils and is working on the Eclipse based tooling for BlackBerry 10. Follow @dougschaefer on twitter.

Marc Khouzam has been working on CDT and been responsible for its Debug component since 2009. He was just appointed co-lead of the CDT project. Marc works for Ericsson in Montreal and is currently focused on the multicore debugging effort and leading the Multicore Debug Workgroup. Follow @marckhouzam on twitter.

52.3. Can you describe the project team and culture?

With a thriving community of close to 20 active committers (See *People in the CDT project [https://projects.eclipse.org/projects/tools.cdt/who]*), more than 250 contributors, and 1500 contributions over the years, the CDT is a strong and dynamic project. It has been under the leadership of Doug Schaefer for close to 10 years and has grown greatly during that time. Marc Khouzam has recently been appointed co-lead to help Doug in the work required by such a large project.

In the first 5 months of 2014 alone, the CDT has received an average of 22 contributions each month from a total of over 25 different contributors.

The CDT aims to be a very open and transparent open-source project. All communications are done publicly and anyone is welcomed to take part, be it in bugzilla discussions, mailing list exchanges, Gerrit changes, or conference calls. Decisions are taken as a group and for the greater good. We welcome newcomers and are always looking for our next committer. We foster an atmosphere of respect and open-mindedness.

Although the CDT is a mature project, its community continually works at keeping it on the bleeding-edge of tools technology. With that in mind, we welcome comments, suggestions and new ideas that help keep the CDT moving forward.

If it happens that you find us a little unresponsive at times, please remember that it is only because we are very busy; don't take offense and don't hesitate to contact the community again. Showing you care about a problem will only help us to understand its importance.

52.4. Where can a contributor find information how to contribute?

A lot of information about the CDT can be found on its wiki: *CDT wiki [http://wiki.eclipse.org/CDT]*

Specifically, for contributions, the following pages are good starting points: *Contributing to CDT Wiki [https://wiki.eclipse.org/CDT/contributing]* and *CDT git [https://wiki.eclipse.org/CDT/git]*

The cdt-dev mailing list is also a good place to ask questions you could not find the answer to.

The CDT actively uses Bugzilla for bug tracking and specific discussions. A contributor should first open a bug in Bugzilla with a description of the problem that is being addressed. The code contribution should then be pushed in Gerrit for review by the committers.

Whenever possible code contributions should be accompanied with tests. API-breaking changes must be avoided and the PDE API-tooling of Eclipse should be used before making a code submission.

Contributors can feel comfortable using the relevant bug or the *cdt-dev mailing [https://dev.eclipse.org/mailman/listinfo/cdt-dev]* list to inquire about a contribution that has not been reviewed after a couple of weeks.

53
The Tycho project

53.1. Can you describe the target and scope of the Tycho project?

Tycho provides Maven plug-ins for building Eclipse plug-ins/OSGi bundles, features, update sites/p2 repositories and RCP applications.

The project was started in 2009 by Igor Fedorenko and moved to the Eclipse Foundation in 2011.

The original Maven Eclipse build implementation was from Tom Huybrechts. Tom chose name *Tycho*. The code then was open sourced at codehaus.org and this is where Igor picked it up under much pressure (quote from Igor) from Jason van Zyl.

53.2. Who are you and how are you involved in the Eclipse community?

My name is Jan Sievers, project co-lead of the Tycho project. I work at SAP since 2001, currently with a focus on build and provisioning and I'm involved with the eclipse community as a "corporate consumer" since 2003 and as project committer since 2011.

53.3. Can you describe the project team and culture?

Tycho has three co-project leads: Jason Van Zyl (Takari), Igor Fedorenko (Takari) and Jan Sievers (SAP)

The Tycho project team consists of 4 committers from 3 companies (SAP, Takari and Bachmann Electronic GmbH). Takari was formerly known as Sonatype.

- Igor Fedorenko (Takari)
- Tobias Oberlies (SAP)
- Jan Sievers (SAP)
- Martin Schreiber (Bachmann Electronic)

We get about 4-5 code contributions per Tycho release from various people. Examples are Mickael Istria, Pascal Rapicault, Alexander Kurtakov.

The project communication should be transparent on mailing lists and Bugzilla. We try to release often (every 3-4 months) and have avoided project graduation due to additional paperwork up to now (formally Tycho is still in incubation although it is already very stable).

We encourage that one change should do one thing (to ease review). Every change goes through the Gerrit code review system and we have an extensive test suite executed for each Gerrit change (both unit and integration tests). We keep this test suite growing to ensure we don't introduce regressions.

We avoid changes without a Bugzilla entry and test case (except trivial or pure refactoring).

Last but not least we recently invested to attract more contributions and eventually committers (contributor guide, giving credit for patches, enhance README files, etc).

Our primary discussion channel is the tycho-user and tycho-dev mailing lists. Some committers also use Twitter to announce Tycho releases, or answer questions on Stack Overflow.

53.4. Where can a contributor find information how to contribute?

See the *Tycho contributor guide [https://wiki.eclipse.org/Tycho/Contributor_Guide]*. In the contributor guide, we advise to ask on tycho-user or bugzilla first. We could use bugzilla tags like *helpwanted* or similar for easy to solve bugs but we don't do this consistently up to now.

This information how to run unit tests is available on the *Developing Tycho Wiki [https://wiki.eclipse.org/Developing_Tycho]*.

54
The Standard Widget Toolkit (SWT) project

54.1. Can you describe the target and scope of the SWT project?

SWT is an open source widget toolkit for Java designed to provide efficient, portable access to the user-interface facilities of the operating systems on which it is implemented. It aims to provide portable Java API to develop UI applications that have the native look and feel.

SWT supports a wide range of operating systems, windowing systems and architectures. It is implemented for Windows - win32, WPF; Linux-Motif, GTK2, GTk3; Mac-Cocoa, Mac-Carbon; Photon. The currently active ports are Windows, Linux GTK2 / GTK3, Mac-Cocoa.

54.2. Who are you and how are you involved in the Eclipse community?

I'm Lakshmi Priya Shanmugam, one of the co-leads of SWT. I work for IBM India and have been a committer in the SWT project for the past 5 years. I primarily work on the SWT Mac-Cocoa port. I've also briefly contributed to the Orion project.

54.3. Can you describe the project team and culture?

SWT currently has 5 active committers and 2 developers. One committer is from RedHat and the others are from IBM.

Bugzilla is the main channel of communication within the team and with the community. All the bug related discussion happens over there. We have recently started using Gerrit for code review and accept contributions from the community through Gerrit.

The SWT user community is very active. Every release, we get contributions from the community in the form of bugs, test cases, patches for enhancements and bug fixes. And of course the team is open to the community participation.

54.4. Where can a contributor find information how to contribute?

The *How to fix a bug in SWT [http://eclipse.org/swt/fixbugs.php]* link is the place to start. It provides useful information including the steps to setup the development environment, links to articles, Bugzilla and operating system documentation.

The *project home page [http://eclipse.org/swt/]* is also a great place to find useful resources and links to articles, downloads, frequently asked questions lists, etc.

The SWT newsgroup is a very active user discussion and help forum. See the Eclipse Community Forums page (expand Eclipse Projects to see SWT) for information on this. SWT development is discussed and tracked in the *Eclipse bugzilla [https://bugs.eclipse.org/bugs/]* under the Platform/SWT component. You can subscribe to the new bug inbox by watching platform-swt-inbox@eclipse.org from your Bugzilla email preferences. If you are modifying or porting SWT and have questions, try the *SWT developer mailing list [http://dev.eclipse.org/mailman/listinfo/platform-swt-dev]*.

When someone wants to contribute a new feature or API, he should first open a bug in Bugzilla and start a discussion to involve the team and community. A contributor can submit his contribution as a patch through the Gerrit review system for review. It is always good to provide a test case or test snippet to test the patch.

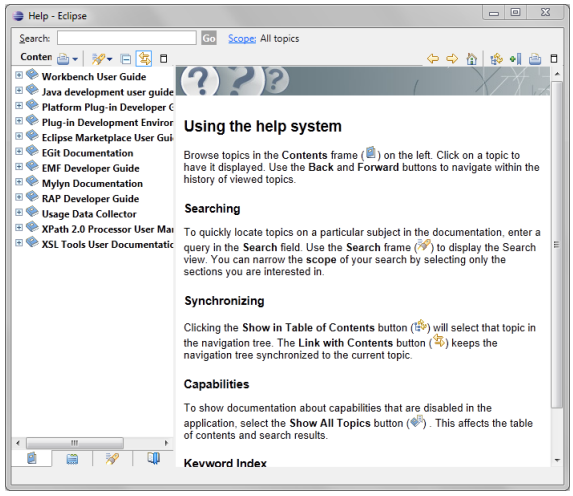
Appendix A. Additional Eclipse resources

A.1. Eclipse online resources

A.1.1. Online documentations

The Eclipse help system is available from within your Eclipse installation as well as online.

With your running Eclipse IDE you can access the online help via *Help → Help Contents*. This will start a new window which shows you the help topics for your currently installed components.



You find the online help for the current release of the Eclipse IDE under the following URL: *Eclipse online help [http://help.eclipse.org]*. The online help is version-dependent and contains the help for all Eclipse projects of the simultaneous release.

A.1.2. Web resources

The Eclipse webpage also contains a list of relevant resources about Eclipse and Eclipse programming. You find these resources under the following link: *Eclipse resources [http://www.eclipse.org/resources/]* and *Eclipse corner wiki [https://wiki.eclipse.org/Eclipse_Corner]*.

You also find lots of tutorials about the usage of the Eclipse IDE from the vogella GmbH on the following webpage: *vogella Eclipse IDE tutorials [http://www.vogella.com/tutorials/eclipseide.html]*.

Information about Eclipse plug-in and RCP development from the vogella GmbH can be found on the following webpage: *Eclipse Plug-in and RCP tutorials [http://www.vogella.com/tutorials/eclipse.html]*.

A.2. Eclipse Rich Client Platform (RCP)

It is possible to run the Eclipse programming model to create native stand-alone applications called Eclipse RCP applications.

To learn about this, see the *Eclipse Rich Client Platform [http://www.vogella.com/books/eclipsercp.html]* book or the *Eclipse plug-in tutorial [http://www.vogella.com/tutorials/eclipse.html]* website.

A.3. More books from the vogella book series

You find more about the books published by vogella under *vogella books [http://www.vogella.com/books/index.html]*.

Appendix B. Additional information

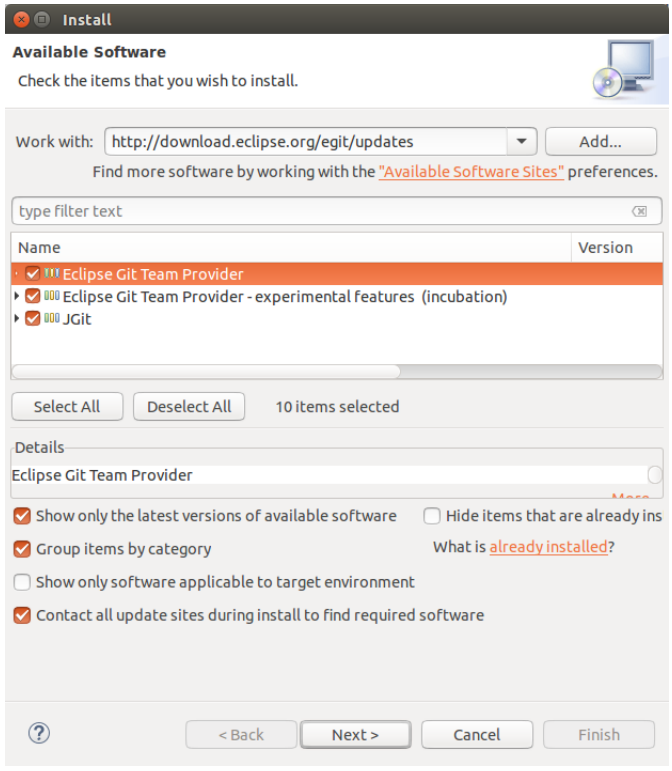
B.1. Installation of Git support into Eclipse

Most Eclipse IDE downloads from Eclipse.org contain support for Git in their default configuration. In this case no additional installation is required.

If the Git functionality is missing in your Eclipse IDE installation, you can install it via the Eclipse installation manager. Start this manager via the *Help* → *Install new Software...* menu entry by using the following update site URL:

```
http://download.eclipse.org/egit/updates
```

The dialog to install the Eclipse Git team provider is depicted in the following screenshot.



B.2. How to configure the usage of Git in Eclipse

B.2.1. Interoperability of Git command line settings with the Eclipse IDE

The Git functionality in the Eclipse IDE uses the same configuration files as the Git command line tools. This makes it easier to use the Eclipse Git tooling and the command line tooling for Git interchangeable.

One notable exception is currently the support of gitattributes. See *Bug 342372 - support gitattributes [https://bugs.eclipse.org/bugs/show_bug.cgi?id=342372]* for details.

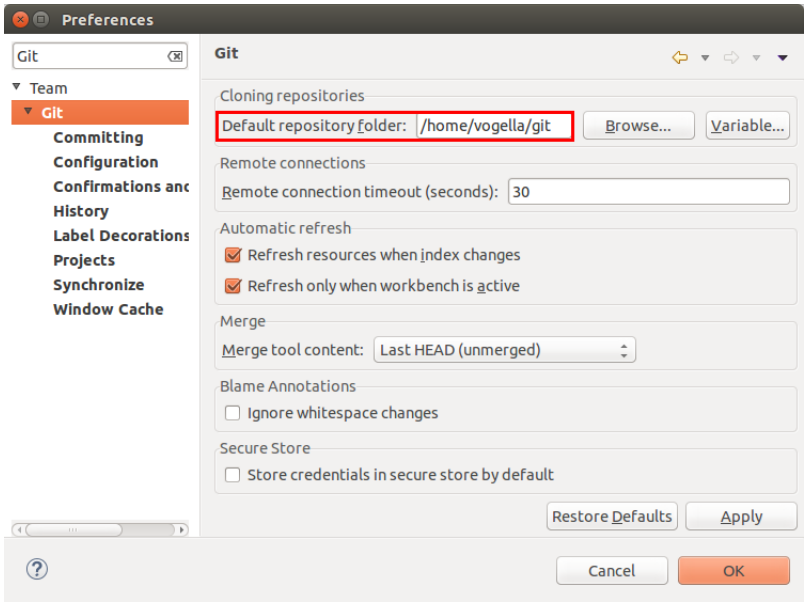
B.2.2. Git user settings in Eclipse

To use Git you must configure your full name and email address. This information is used to fill the author and committer information of commits you create. These Git configuration settings can be adjusted via the Eclipse preference setting. Select *Window* → *Preferences* → *Team* → *Git* → *Configuration* to see the current configuration and to change it.

You can add entries to your Git configuration by pressing the *Add Entries* button on the *Git Configuration* preference page.

B.2.3. Default clone location

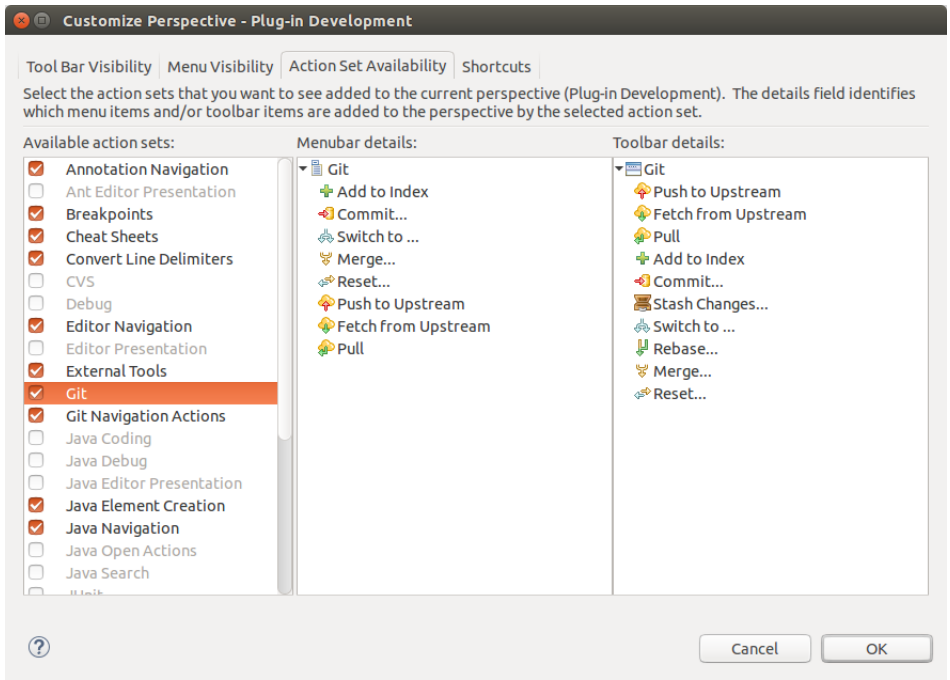
If you clone a new repository via Eclipse Git, it will create by default a new sub-folder for the new Git repository in a default directory. This default path can be configured via the *Windows* → *Preferences* → *Team* → *Git* entry in the *Default Repository folder* field.



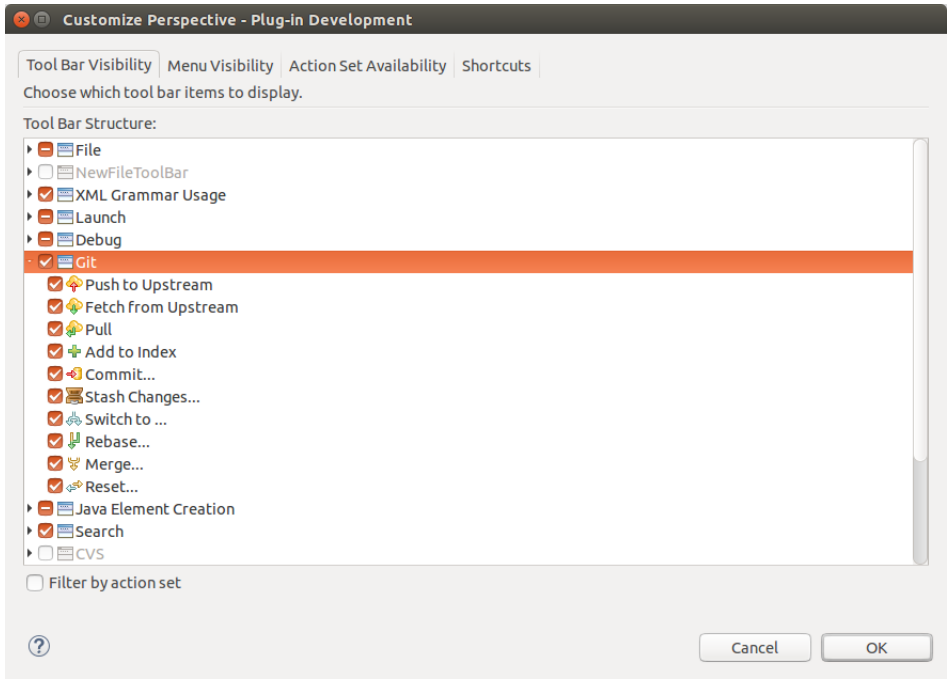
You can also use Eclipse configuration variables to define this path, e.g., if you want to store repositories in the folder "git" under the Eclipse workspace you may use ${workspace_loc}/git.

B.2.4. Configuring the toolbar and the menu for Git usage

To simplify access to the common Git operations you can activate the Git toolbar. For this select *Window+Perspective* → *Customize perspective...* and check the *Git* and *Git Navigation Actions* entries in the *Action Set Availability* tab.



Afterwards you can configure which Git operations should be available via the *Tool Bar Visibility* or the *Menu Visibility* tab.

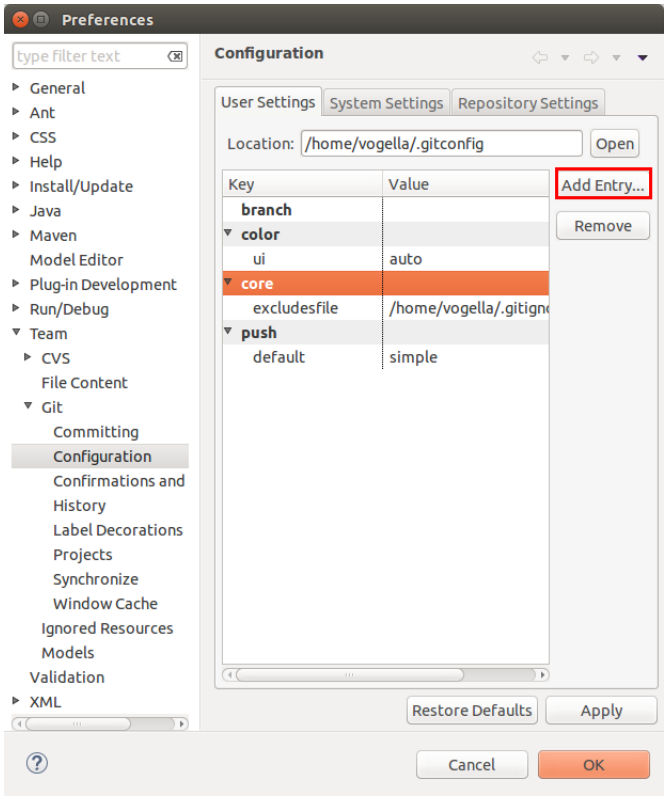


B.3. Git user configuration for the Eclipse IDE

B.3.1. Validate your Git user settings

Select *Window* → *Preferences* → *Team* → *Git* → *Configuration* and add your full name and email setting to your user settings if this is not yet configured.

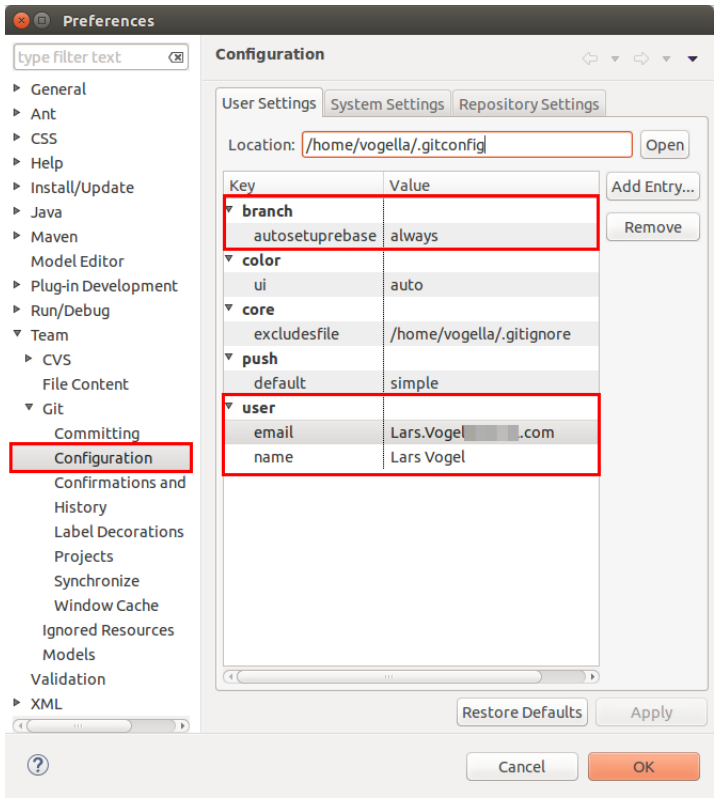
The user should be stored under the `user.name` key with the full name as value. If it is not available press the *Add Entry...* button. Repeat the procedure for your email address via the `user.email` key.



B.3.2. Configure Git to rebase during pull operations

Unless you have a different opinion about rebase and merge commits, also set the `branch.autosetuprebase` parameter to `always`. This is common setting for Git repositories to avoid merge commits if you pull from a remote repository and have divergent changes and instead rebases your local branch on the remote branch it tracks.

After this setup, the configuration should look similar to the following screenshot.



B.4. Authentication via SSH

B.4.1. The concept of SSH

Most Git (and Gerrit) servers support SSH based authentication. This requires a *SSH key pair* for automatic authentication.

An SSH key par consists of a public and private key. The public key is uploaded to the application you want to authenticate with. The application has no access to the private key. If you interact with the hosting provider via the ssh protocol, the public key is used to identify a user who encrypted the data during communication with the corresponding private key.

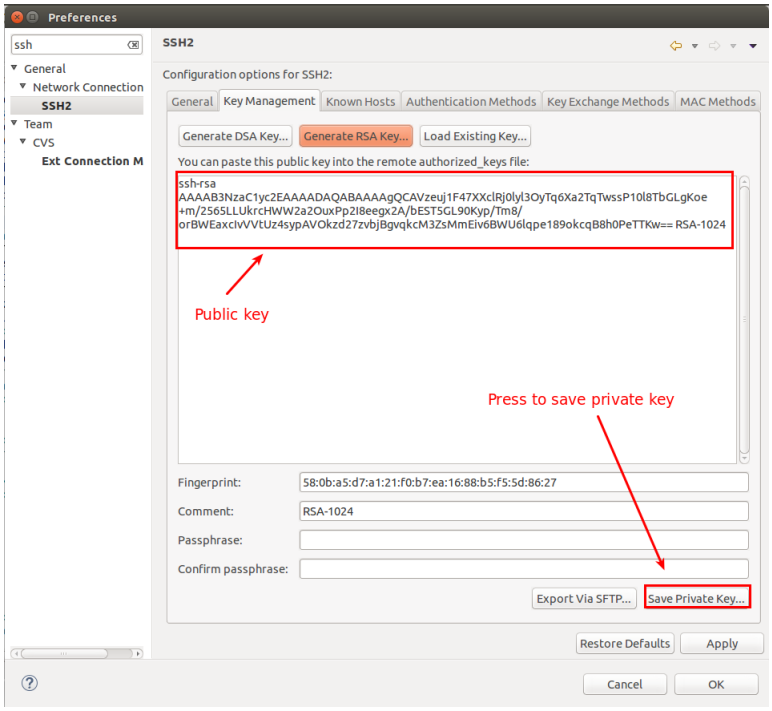
B.4.2. SSH key pair generation

To create an SSH key under Linux (or Windows / Mac with OpenSSH installed) switch to the command line and execute the following commands. The generated SSH key is by default located in the `.ssh` directory of the user home directory. Ensure that you backup existing keys in this directory before running the following commands.

```
# Switch to your .ssh directory
cd ~/.ssh
```

```
# If the directory
# does not exist, create it via:
# mkdir .ssh

# Manually backup all existing content of this dir!!!

# Afterwards generate the ssh key
ssh-keygen -t rsa -C "your_email@youremail.com"

# Press enter to select the default directory
# You will be prompted for an optional passphrase
# A passphrase protects your private key
# but you  have to enter it manually during ssh operations
```

The Eclipse IDE allows you to create an SSH key pair via *Window* → *Preferences* → *General* → *Network Connection* → *SSH2*.



It is good practice to use a passphrase to protect your private key. It is also good practice to use operating system level permission settings to ensure that only the owning user can access the `~/.ssh` folder and its content.

Note

In the above `ssh-keygen` command the -C parameter is a comment. Using your email is good practice so that someone looking at your public key can contact you in case they have

questions. Including the email enables system administrators to contact the person in case of questions.

The result will be two files, `id_rsa` which is your private key and `id_rsa.pub` which is your public key.

You find more details for the generation of an SSH key on the following webpages: *GitHub Help: description of SSH key creation [https://help.github.com/articles/generating-ssh-keys]* or *OpenSSH manual [http://www.openssh.com/manual.html]*.

Tip

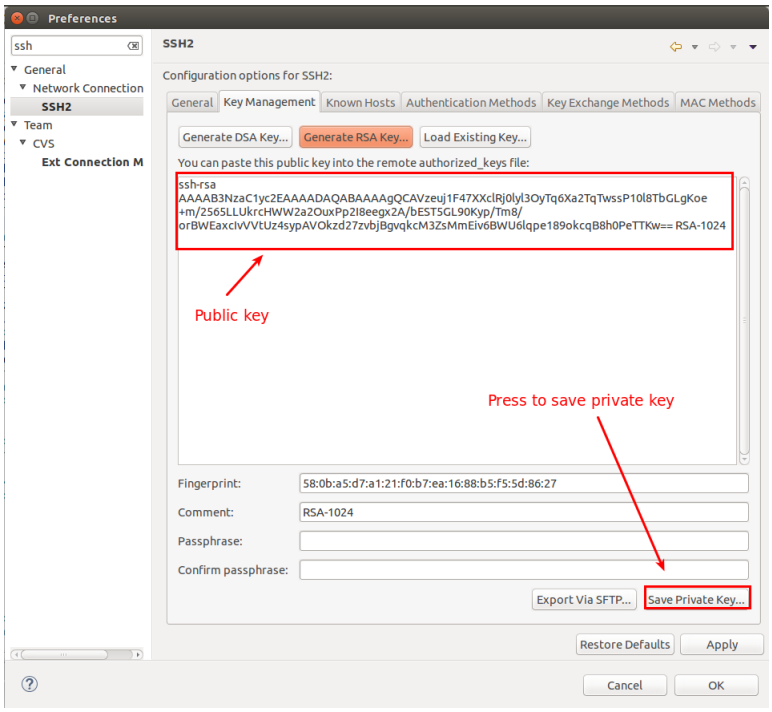
You can specify alternative key names with the `-f` parameter on the command line. This is helpful if you have multiple different repositories and you want to have a different key for each one. For example, you can name your SSH keys in domain name format, e.g., `eclipse.org` and `eclipse.org.pub` as well as `github.com` and `github.com.pub`.

You need additional configuration in the `.ssh/config` file, because only the `id_rsa` will be picked up by default. The following code shows an example.

```
Host *.eclipse.org
  IdentityFile ~/.ssh/eclipse.org

Host *.github.com
  IdentityFile ~/.ssh/github.com
```

B.4.3. Eclipse support for SSH based authentication

The Eclipse IDE allows you to create an SSH key pair for SSH based communication via *Window* → *Preferences* → *General* → *Network Connection* → *SSH2*.



B.5. Appendix: Cloning from the Git server and adjusting the push URL

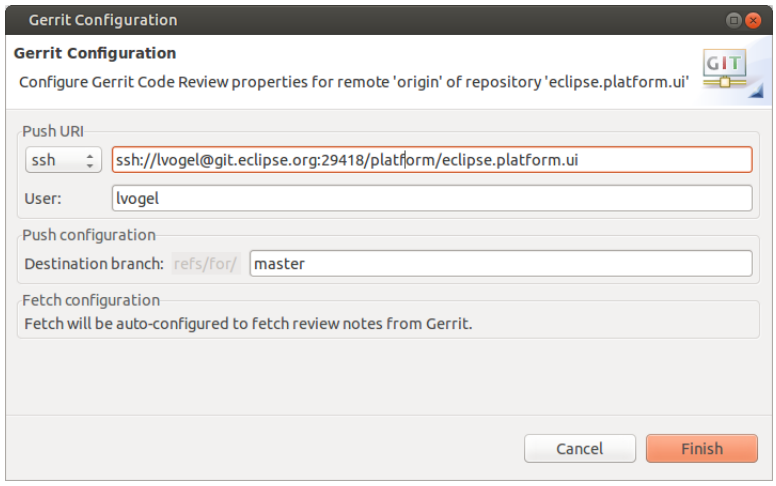
B.5.1. Overview

If you cloned an Eclipse Git repository directly from the Eclipse Git server (without using the Gerrit server) you have to adjust the push URL to create Gerrit changes. For example, you may have found the clone URL on: *Eclipse Git web interface [https://git.eclipse.org/c/]*.

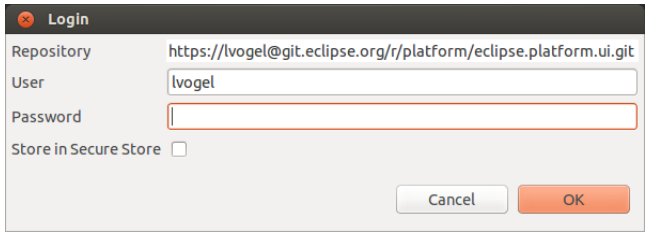
It is far easier to clone from the Gerrit server as this does not require you to change the push URL. See Section 13.1, “Finding the correct Git repository” for the description how to clone. In case you can use the Gerrit server, this section is not relevant for you.

The push configuration in the following dialog depends a bit if you want to use SSH or HTTPS. If you want to use SSH ensure to use the 29418 port and remove the "gitroot" string from the push-url.

The following screenshot demonstrates that for the Eclipse platform UI Git repository.



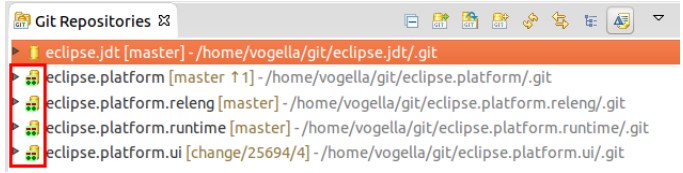
If you want to use HTTPS to push to the Gerrit server, you have to use a push URL with an "r" included (For example https://userid@git.eclipse.org/r/platform/eclipse.platform.ui.git). You need to provide in this case your Gerrit password if you want to push to an Eclipse Git repository as depicted in the following screenshot.



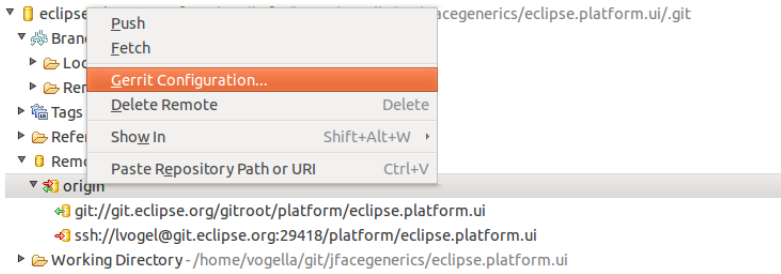
B.5.2. Gerrit push configuration

The Gerrit server requires that you push using a predefined *refspec*, called *HEAD:refs/for/master*. A refspec allows you to configure which remote branch should be used for remote operations.

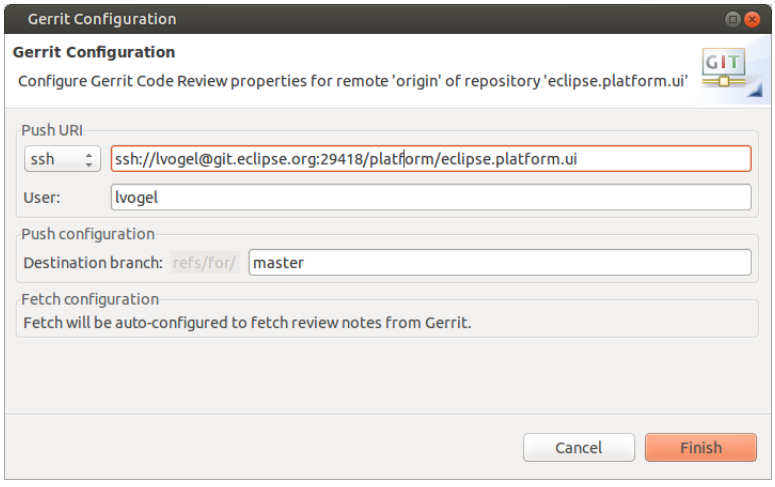
If you clone a Git repository managed by a Gerrit server, this push url is already correctly configured in most cases. The icon for repository configured to be used for Gerrit uses a green icon. Also Gerrit specific commands are added to the repository's context menu, e.g., *Push to Gerrit...* and *Fetch from Gerrit...*). In addition, the repository is configured to always add a Change-ID to the commit message. In the following screenshot the repositories configured for Gerrit are highlighted. The `eclipse.jdt` repository in this screenshot is not configured for Gerrit.



If you have to configure the push URL manually, select your remote repository in the *Git Repositories* view, right-click on it and select *Gerrit configuration*.



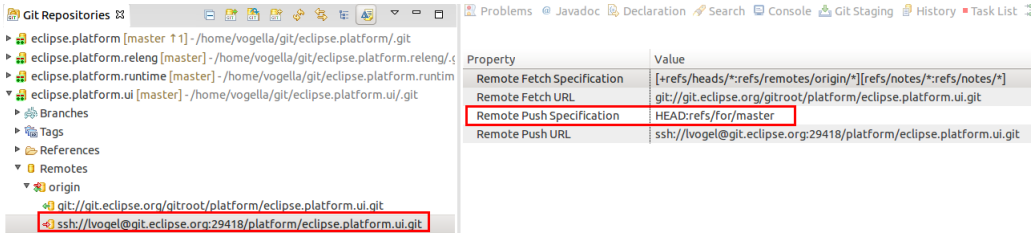
If you select the *origin* entry, right mouse click on it and select *Gerrit Configuration...* the entry should look like the following screenshot.



Note

For HTTPS access the URL is different.

You can validate the push specification if you select the highlighted node in the following screenshot and check the *Remote Push Specification* entry in the *Properties* view.

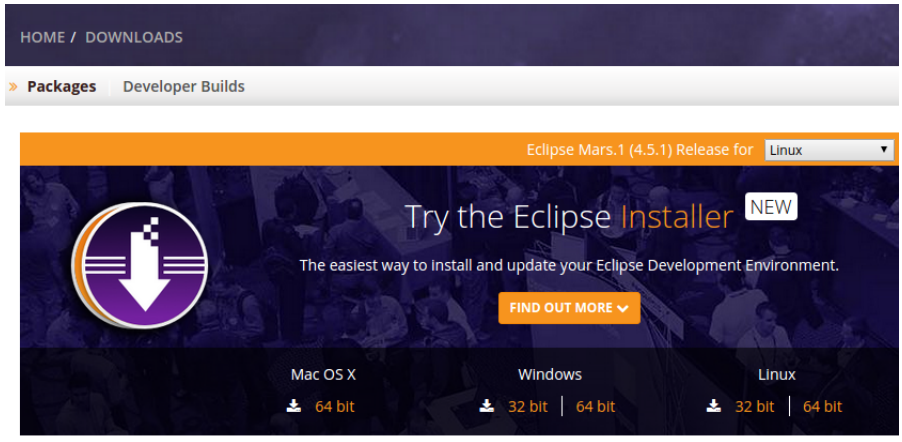


B.6. Using the Eclipse installer for code contributions

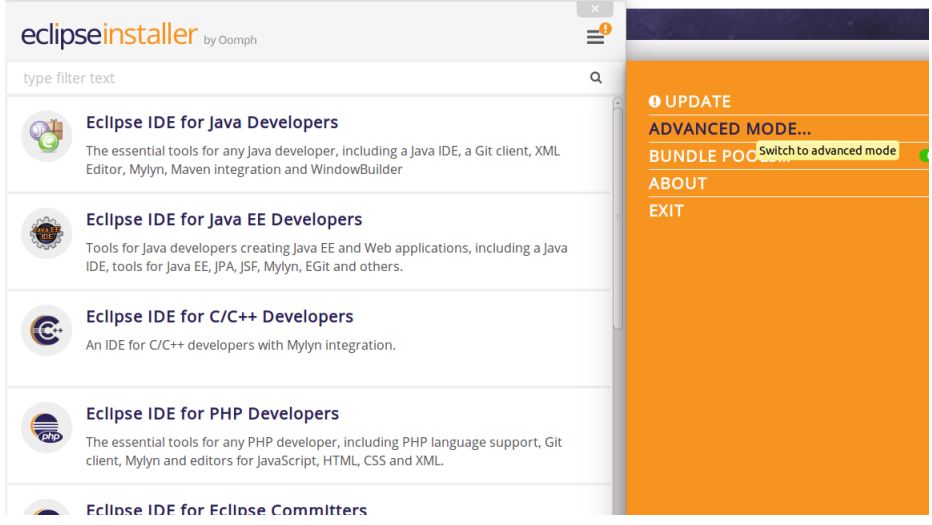
You can also use the Eclipse installer (which is provided by the Eclipse Oomph project) to automated some of the steps of code contributions. Please note that the Eclipse installer is currently still under heavy development and the user interface might change significantly.

> **Note**
>
> Currently the main Eclipse platform projects do not actively configure Oomph, hence the contribution setup via the Eclipse installer may not result in a correct setup.

Download for this, the Eclipse installer from *Eclipse installer [https://www.eclipse.org/downloads/]*. Depending your operating system you have to extract it, or you can just run the executable.

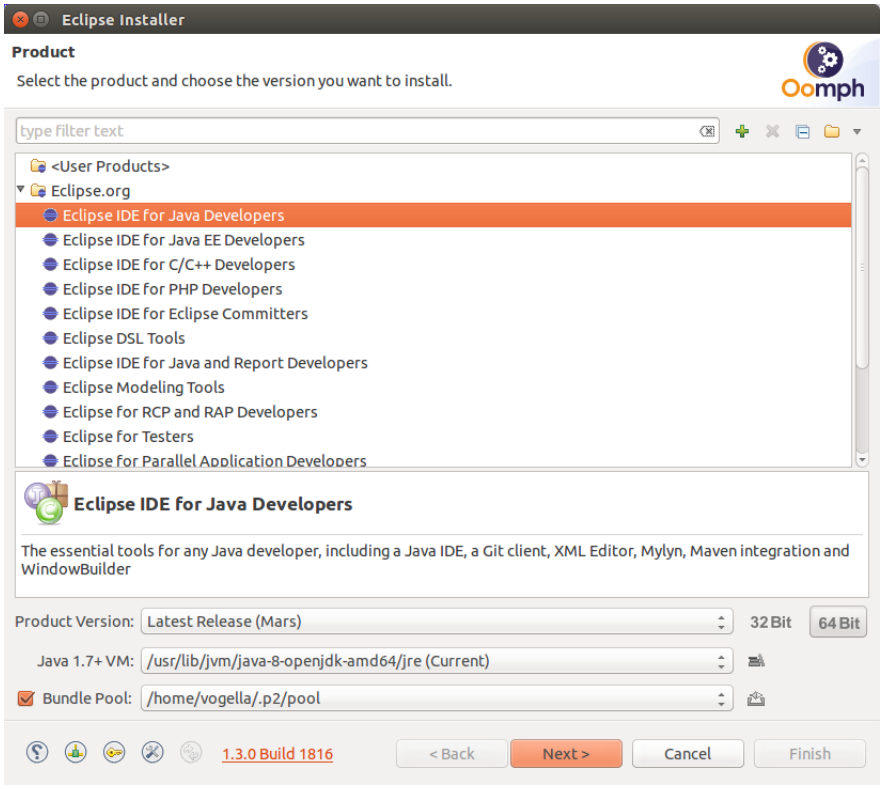


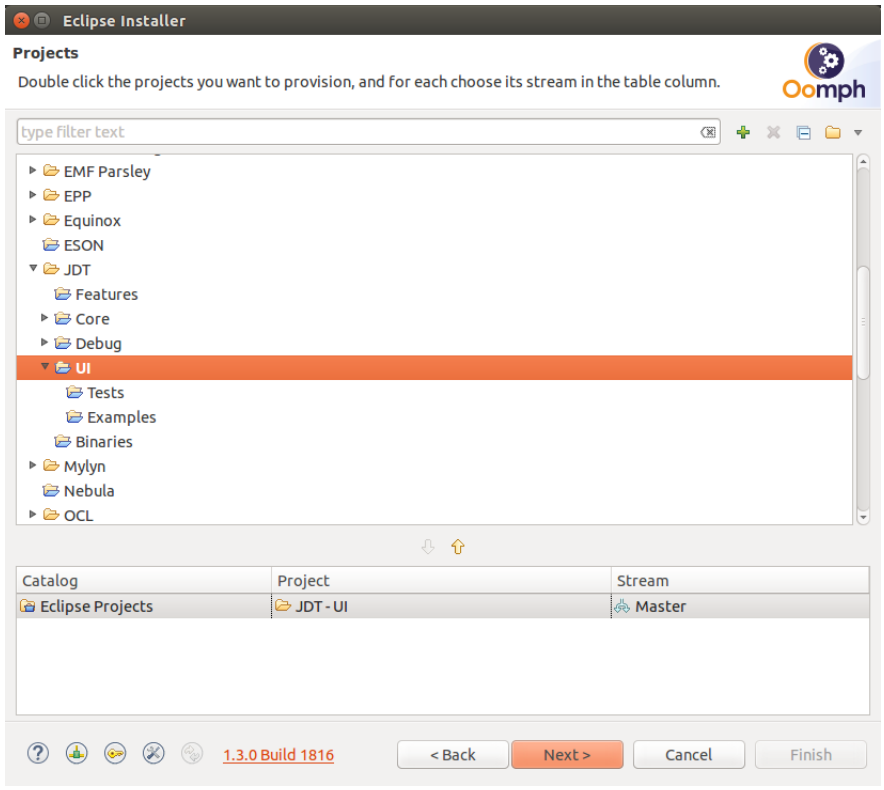
Start it and select the *Advanced Mode*.

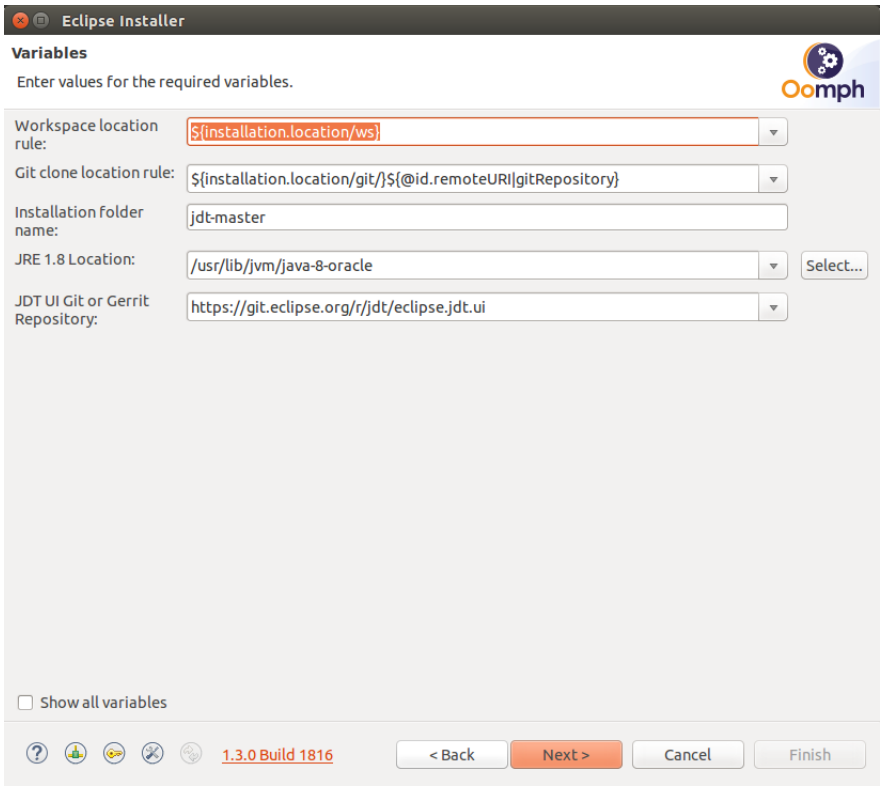


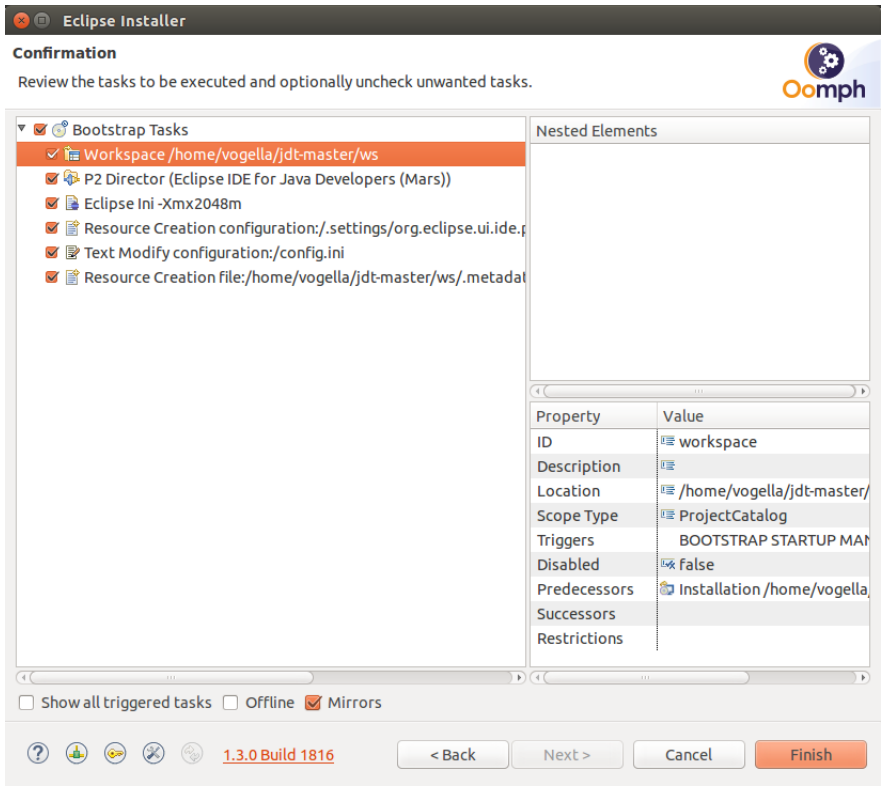
Select *Eclipse IDE for Java Developers*.

Afterwards you can select the project you want to contribute to. If this project maintains a correct Oomph configuration will clone the corresponding repository and configure your workspace. The following screenshots are from JDT UI, for which the Eclipse installer configuration is valid as the time of this writing.

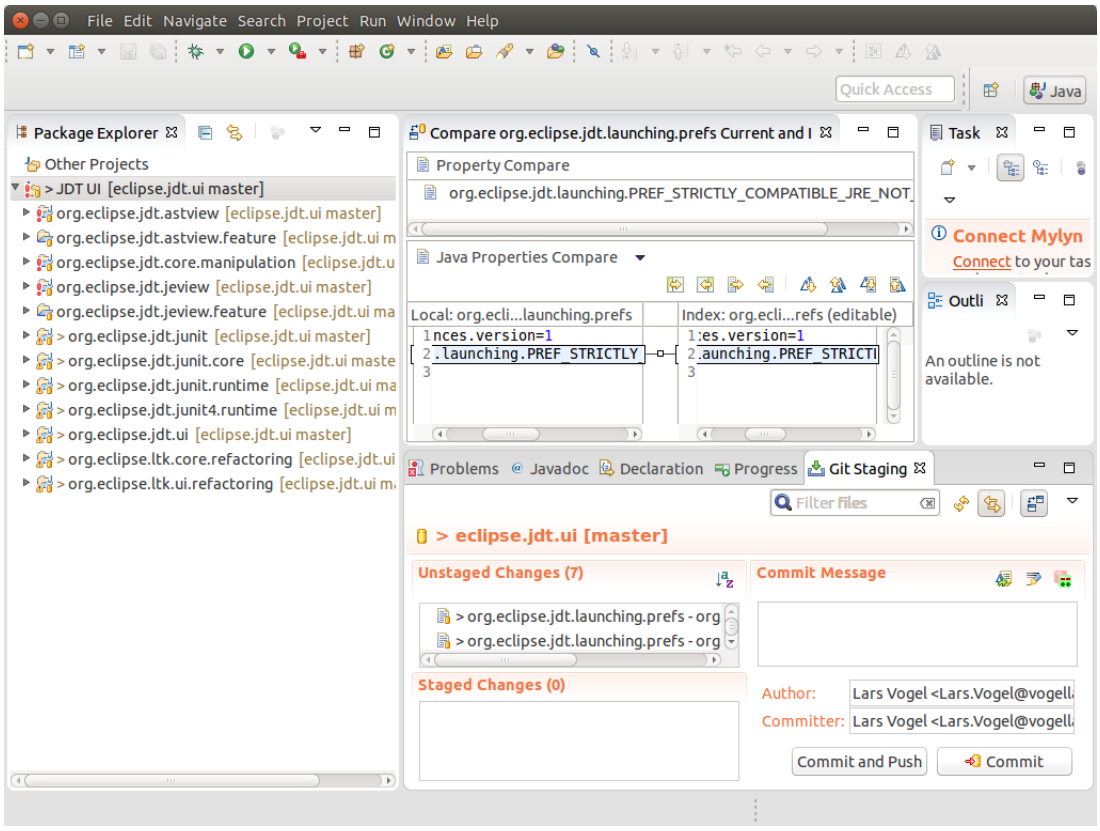



Eclipse Installer
Projects
Double click the projects you want to provision, and for each choose its stream in the table column.
Oomph
type filter text
EMF Parsley
EPP
Equinox
ESON
JDT
Features
Core
Debug
UI
Tests
Examples
Binaries
Mylyn
Nebula
OCL
Catalog
Project
Stream
Eclipse Projects
JDT - UI
Master
1.3.0 Build 1816
< Back
Next >
Cancel
Finish


Eclipse Installer
Variables
Enter values for the required variables.
Oomph
Workspace location rule:
${installation.location/ws}
Git clone location rule:
${installation.location/git/}${@id.remoteURI|gitRepository}
Installation folder name:
jdt-master
JRE 1.8 Location:
/usr/lib/jvm/java-8-oracle
Select...
JDT UI Git or Gerrit Repository:
https://git.eclipse.org/r/jdt/eclipse.jdt.ui
Show all variables
1.3.0 Build 1816
< Back
Next >
Cancel
Finish



You end up with a workspace with the existing Git repositories. After this initial setup you should consult the contribution guide of the project to ensure you know their contribution guidelines.


File Edit Navigate Search Project Run Window Help
Quick Access
Java
Package Explorer
Other Projects
> JDT UI [eclipse.jdt.ui master]
org.eclipse.jdt.astview [eclipse.jdt.ui master]
org.eclipse.jdt.astview.feature [eclipse.jdt.ui m
org.eclipse.jdt.core.manipulation [eclipse.jdt.u
org.eclipse.jdt.jeview [eclipse.jdt.ui master]
org.eclipse.jdt.jeview.feature [eclipse.jdt.ui ma
> org.eclipse.jdt.junit [eclipse.jdt.ui master]
> org.eclipse.jdt.junit.core [eclipse.jdt.ui maste
> org.eclipse.jdt.junit.runtime [eclipse.jdt.ui ma
> org.eclipse.jdt.junit4.runtime [eclipse.jdt.ui m
> org.eclipse.jdt.ui [eclipse.jdt.ui master]
> org.eclipse.ltk.core.refactoring [eclipse.jdt.ui
> org.eclipse.ltk.ui.refactoring [eclipse.jdt.ui ma
Compare org.eclipse.jdt.launching.prefs Current and I
Property Compare
org.eclipse.jdt.launching.PREF_STRICTLY_COMPATIBLE_JRE_NOT
Java Properties Compare
Local: org.ecli...launching.prefs
Index: org.ecli...refs (editable)
nces.version=1
.launching.PREF_STRICTLY
es.version=1
.aunching.PREF_STRICTI
Task
Connect Mylyn
Connect to your tas
Outli
An outline is not available.
Problems
Javadoc
Declaration
Progress
Git Staging
Filter files
> eclipse.jdt.ui [master]
Unstaged Changes (7)
> org.eclipse.jdt.launching.prefs - org
> org.eclipse.jdt.launching.prefs - org
Staged Changes (0)
Commit Message
Author:
Lars Vogel <Lars.Vogel@vogell
Committer:
Lars Vogel <Lars.Vogel@vogell
Commit and Push
Commit

Index

A

Advantages of code review, 49
Application under test, 81

C

CDT project, 183
Code review, 49
Code under test, 81
Committer, 13
Configuration files of plug-ins, 100
Contributor, 13
Copyright header update , 159
Copyright header update tool , 159

D

Dashboard, 17
Deployment of plug-ins, 123
Deployment using update sites, 125
Dropins folder, 124

E

Eclipse Architecture, 97
Eclipse core components, 98
Eclipse dashboard, 17
Eclipse Foundation, 5
Eclipse Git
    preference settings , 194
    toolbar , 194
Eclipse high level design, 97
Eclipse IDE
    custom build, 163
    Downloads, 11
Eclipse IDE for Java development, 11
Eclipse open source project, 3
Eclipse Public License, 9
Eclipse RCP, 19
    Book, 191
Eclipse releases, 3
Eclipse Rich Client Platform
    Book, 192
Eclipse SDK, 25
Eclipse sub-project, 7
Eclipse top-level projects, 7
EGit project, 179
EPL, 9

F

Feature patch projects, 137
Feature projects, 121
Features, 121
Find the plug-in for a certain class, 149
Functional test, 81

G

Gerrit
    Change , 51
    definition, 51

H

History, 19

I

Include all plug-ins in Java search, 149
Installation
    e4 tools, 143
Installation of Git support into Eclipse, 193
Integration test, 81
Internal API, 98

J

Java version requirements to run Eclipse, 95
JDT, 25
JDT project, 177
JUnit
    @Test, 83
    Annotations, 87
    Assert statements, 87
    Example test method, 83
    Test execution order, 88
    Test suite, 84
JUnit framework, 83

L

Lars Vogel - Author , xiv
Live model editor (see Model spy)

M

m2e, 181
MANIFEST.MF, 100
Maven in Eclipse, 181
Menu spy, 141
Model spy, 145

N

Naming conventions for test methods, 84

O

Online help for Eclipse, 191
Open Plug-in Artifact, 150

P

p2 update site, 125
PDE, 25
Performance test, 81
Plug-in, 19
Plug-in deployment, 123
    export and installation into dropins folder , 124
    installation into host IDE , 123
    Using update sites , 125
Plug-in development, 19
Plug-in search, 151
Plug-in Spy, 141
plugin.xml, 100
Project pages, 15
Provisional API (see Internal API)

R

Run arguments (see Run configuration arguments)
Run configuration, 103
Run configuration arguments, 104
    clearPersistedState, 104
    console, 104
    consoleLog, 104
    nl, 104
    noExit, 104
Runtime Eclipse IDE , 101

S

SDK, 25
squash commits first, 67
SSH key pair, 197
Starting the IDE from the Eclipse IDE , 101
SWT project, 189

T

Test annotations from JUnit, 87
Test fixture, 81
Text search, 152
Tracing in Eclipse, 155
Tycho project, 187

U

Unit test, 81
Update sites, 125
User interface guidelines for Eclipse, 13

V

vogella books, 192
vogella GmbH, xiv

W

Web resources for Eclipse, 191