DPF-Editor: Developer Guide-Lines 0.2

Florian Mantz

April 11, 2012

Abstract

This document should give some help to work with the DPF workbench project. In addition, it should guide to build stable releases. This document may be updated in the future.

1 Project-Overview

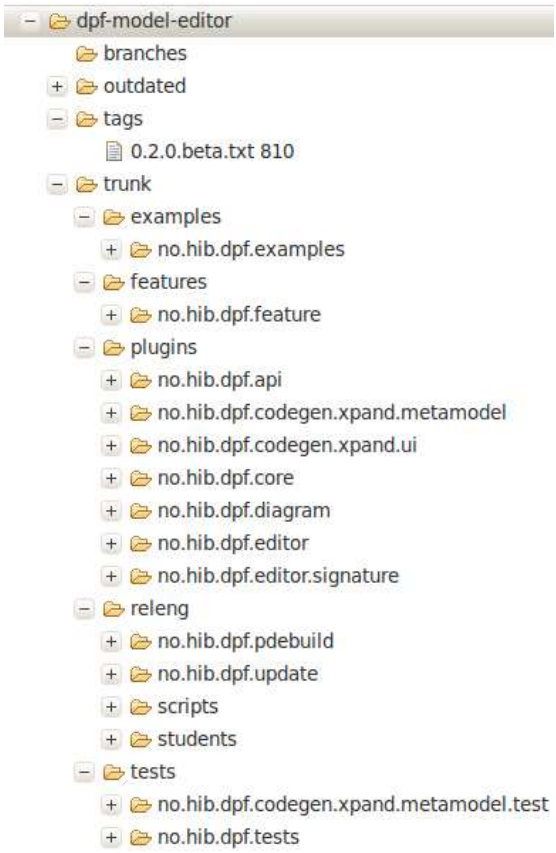

Figure 1: DPF-Editor project structure

The current project structure follows mainly guidelines that are also used in "official" eclipse projects (`http://dev.eclipse.org`). This section will give an overview over the used project structure. The current root node for the project is "dpf-model-editor".

branches: Current branches are located in the branches directory. At the moment this directory is empty. However, if there should be in the future any branch, it will be located in a subdirectory of the branch directory.

outdated: The directory "outdated" is a "trash" directory. We created it only because we merged earlier branches and wanted to have an easy access to the history of the files. It may be the case that we will delete the whole directory in the future.

tags: The directory "tags" will contain text-files identifying versions of the tool. A file like "0.2.0.**beta**.txt" indicates a test version. After the version is finally tested and hopefully bug-free, we will create a version e.g. "0.2.0" by adding a new file "0.2.0.txt". However, if some bugs are fixed that had not been found earlier it may be the case to fix them in updated versions e.g "0.2.1" (new text-file). Note, that a tagged version conceptually does not start an own branch since it does not start an own history. However, Subversion, does not distinguish between "tag"s and "branch"s. Therefore we decided to use text files as "tags". You checkout then a specific version of the tool by checking out the projects from the trunk directory using the revision number specified in the text file. The advantage is that if you have checked out a revision number you can still synchronize with the "head" and get new changes. If you use a "branch" as tag as usually for SVN proposed, you start an own SVN history you synchronize against. Hence, you will not get the changes from the head.

trunk: The trunk directory contains the "head" of the project i.e. the current files. It has several subdirectories which are important for "DPF-developers". More detailed information about what the projects contain you found in the thesis of *Øyvind* and *Anders* on `http://dpf.hib.no/publications/` and `http://gs.hib.no/mediawiki/index.php/EclipsePlugin`. More information can also be found in the source code.

- The subdirectory "examples" contains projects containing examples. At the moment there is only one project "no.hib.dpf.examples" with example specifications. However, there may be in the future also other example projects, e.g. with code generation examples. Note, that also these projects may require updates e.g. after there have been changes in the internal data structure.
- The subdirectory "features" contains the feature project of the tool. With the help of the feature project you can export a local "update-site" which you can use afterwards to install the workbench via "Eclispe → Help → Install New Software". At the moment there is only one feature project. However, there may be other projects for optional features in the future.
- The most important subdirectory is the "plugins" directory. It contains all the code of the DPF editor. The directory "no.hib.dpf.api"

contains "some" code that is used in the editor project. It has to be cleaned-up and merged with the editor project in the future. The *eclipse* projects starting with "no.hib.dpf.codegen.xpand" contain the code which provide the code generator functionality. It extends the popular "xpand" template framework (details you find in Anders's master thesis). The project "no.hib.dpf.core" contains the internal EMF metamodel for the DPF project. It contains the code to e.g. create a "model graph". It mainly provides low-level access to the internal data structure of the DPF models. The project "no.hib.dpf.diagram" contain another *EMF* metamodel. The metamodel extends the metamodel of the "no.hib.dpf.core" project by elements that are only used to visualize the "concepts" of the "core" project. The project "no.hib.dpf.editor" contains the code that implements the visual editor where you can create your own models and metamodels. It makes use of the earlier mentioned projects. The project "no.hib.dpf.editor.signature" contains the component which extends the model editor via an editor to create own signatures (the editor to define own predicates and their semantics ("constrains")).

- The subdirectory "tests" contains the test projects for the DPF workbench. In particular, project "no.hib.dpf.test" contains the JUnit test-cases for the projects "no.hib.dpf.core" and "no.hib.dpf.diagram". Note, the projects contain "Test-Suites" which can be executed as "JUnit-plugin tests".

releng: The releng directory is a directory that contains "meta" information and projects for the "DPF project" like this document. In particular it contains a project "no.hib.dpf.pdebuild" for automatic project builds on the server with the PDE build tool. And furthermore a project that is required to build the "official update-site". The script folder contains some management scripts for managing the DPF project. However, currently it is basically empty. The folder "students" contain some information on what workbench features currently students work and which projects they are going to "touch". It mainly should help that students know with whom they need to talk and discuss when they went to add changes to the project.

2 The Ten Commandments for DPF Developers

[1]

1. **Talk with your colleagues and organize your development that it fits in the global setting of the project.** The DPF workbench project shall become one product even it consists of several components developed by different people.
2. **Be critical and start to think by your own.** If you are developing a component you are the one who will spend the most time thinking about this task. Hence, think about how to solve the problem in an elegant and useful way. Try to use existing technology for tasks where a

[1] partly inspired by `http://zeeshansohail.blogspot.com/2009/01/ten-commandments-for-programming.html`

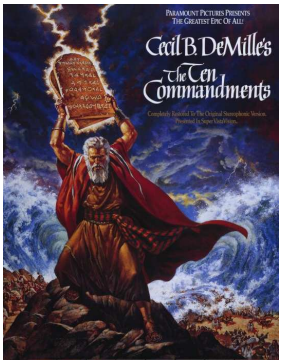

Figure 2: The ten commandments

it fits well (e.g. in most cases it does not make sense to write an own XML parser). Make an own plan how you want to solve your task. Start with an easy design and elaborate it with the requirements. Do not start coding before you know what you are doing. Discuss with your colleges and supervisors what you are going to do. Inform also your colleagues on which projects you are planning changes by keeping the text file "/trunk/releng/students/workingList.txt" up to date. Add a short comment to the file, also when you finished with some changes on one project.

3. **Test your code.** Develop automatic tests using JUnit for important functionalities. Test your code on windows and linux for a new release. If a bug appear, think about why there is not a automatic test case that detected the bug earlier. You do not need to install a separate operating system. You can use a VM e.g the tool VirtualBox[2].

4. **Consider those that come after you.** Provide enough documentation and comments that your code is understandable for your colleagues. Think of the guy that has to maintain your code in the future, as a homicidal psychopath, that knows where you live. Write your code in a way that would make sure he never gets the urge to seek you out.

5. **Getting it to work is NOT ENOUGH!** Take pride in what you do. Honestly, just because you finally got your program working, does not mean it is done. Clean up your code, comment it where needed and test the damn thing thoroughly. Refactor your code when it is required.

6. **Try to create loosely coupled components.** Try to use mainly interfaces from other projects and export as less of a projects as required. If recurring extensions have to be developed or you consider "Non-DPF" developers to add own functionality, consider to work with eclipse´s extension points. However, code that belong together should be in the same project. If code is cyclic depended on each other it seldom makes sense to split it into separate projects.

[2] `https://www.virtualbox.org/` and `http://virtualboxes.org/images/`

7. **Do not create a branch in the SVN by your own.** There should be a serious reason to create a branch. Branches should be used (in this project) if the tool diverges into different directions. Hence, it should mainly be used if you want a different tool starting from the DPF tool. Branches should not be used to separate your work from your colleges' work (by now). This means you should basically **not** commit code to the SVN that prevents the tool to function. However, try to continuously contribute your functioning code into the head. There exist other opinions about this. However, we decided to use this approach, so that we will not require big "branch-merging" sessions in the future and developers see early what is going on in the project.

8. **Do not check-in code that does not compile but permanently backup your work.** Eclipse has a local history for your files. However, it is better to create an backup of your work using a second version control system on your local machine which is not **SVN**. Use preferable **git** for this task. To work with a local git repository is not so difficult and you can find many tutorials in the web [3]. There are also a lot of graphical GUIs if you prefer one. However, commit functionality that is working continuously to the SVN also.

9. **Make sure that your commit is complete.** Commit your code preferable with a single commit and provide a comment to the SVN what you have done. If you are unsure about this: use a new eclipse instance and workspace, checkout your code and check that it compiles and runs.

10. **Respect the user.** Yes, there will be times when you think the user is the most irritating, uneducated, short-sighted being produced by this universe, for the sole purpose of making your life hell. When you respect the user, listen to their opinions and grievances while taking action to improve their lives, you invariably end up with someone, that will listen to you and take your advice.

This rules are not engraved into stone. If you have other opinions about certain points discuss them inside our DPF group. Furthermore, also consider the guidelines from the Wiki[4]. You can create a user account[5] for the wiki yourself and add useful information.

3 How-to Build a New Release

1. Update the plugins version number (to the release number).

2. Fix your code so that all your automatic test cases work and you do not find any bugs anymore when you run the application manually.

3. Make a new eclipse instance as well as a new workspace.

4. Check-out all the projects in "trunk/plugins" and "trunk/tests".

[3] e.g. `http://www.vogella.de/articles/Git/article.html` and `http://nathanj.github.com/gitguide/tour.html`

[4] `http://gs.hib.no/mediawiki/index.php/EclipsePlugin/DeveloperGuidelines`

[5] `http://gs.hib.no/mediawiki/index.php?title=Special:UserLogin&returnto=Special:UserLogin`

5. Run the DPF workbench again and repeat step 2. Fix bugs if necessary. Each developer should be mainly responsible for his own code but also have some responsibility for the code of others. This is necessary since developers leave the project from time to time.

6. If everything works then switch the operating system to windows respectively linux and repeat step 3-5.

7. Create a "tag" with the version number and "beta". (From one of these fresh instances!)

8. Checkout the feature projects and update site project and update the update-site.

9. Install the eclipse workbench now via the update site on linux and windows using fresh eclipse instances.

10. Test that the application is working on windows and linux probably. If not fix the bugs and update the update site again.

11. If everything works fine. Create a new tag in the repository with only the version number.

This versions people can checkout to work with if they wanna try. However, the preferred way for users should be the update site. A new developer should first try to work with the head. Only if this does not work for her/him she/he should checkout a stable version and merge with the head afterwards. In any case this should not be the usual way. At least the classes he/she is working with should be preferable from the head.