Organisation of 3Worlds java projects: technical documentation for 3Worlds developers

Jacques Gignoux – jacques.gignoux@upmc.fr · Ian D. Davies – ian.davies@anu.edu.au – Version 1.6, 15 April 2021 |

Table of Contents

by **Jacques Gignoux** & **Ian D. Davies**

Version: **1.6** (15 April 2021)

Please read this document in full before attempting contributions to 3Worlds.

1. Overview

This document describes how to setup a working environment to contribute to 3Worlds. The 3Worlds codebase is organised as a set of automomous reuseable projects, referred to here as *libraries*. 3Worlds library development follows a particular set of guidelines. We use ivy (http://ant.apache.org/ivy) to handle dependency management and the **eclipse** integrated development environment (IDE). We assume some familiarity with the **eclispe** IDE.

1.1. 3Worlds libraries

Currently, the 3Worlds project has produced the following libraries:

omhtk

omhtk stands for *One More Handy Tool Kit* and is a library of generic, very low-level interfaces (e.g. `Sizeable` for a class which instances have a size, `Resettable` for classes which can be 'reset', etc...) plus very commonly used utilities people keep rewriting all the time (e.g. an euclidian distance function or a time conversion method). Almost all other 3Worlds libraries depend on this one.

omugi

omugi stands for *One More Graph Implementation*. It implements classes to represent dynamic graphs.

uit

uit stands for *Universal Indexing Tree*. It implements classes to to provide efficient searching of spatial data. The base class is an `IndexingTree`. It is a generalisation of a QuadTree (https://en.wikipedia.org/wiki/Quadtree), more accurately called a *k*-d tree (https://en.wikipedia.org/wiki/K-d_tree). It is based on work by **Paavo Toivanen** found here (https://dev.solita.fi/2015/08/06/quad-tree.html).

rvgrid

rvgrid stands for *Rendezvous Grid*. It contains a very basic implementation of ADA (https://www.adaic.org/)'s famous rendezvous system used to exchange data between parallel tasks and an implementation of a universal discrete state machine designed by **Shayne Flint**.

qgraph

qgraph is a *Query system for Graphs*. It implements a Query system that can check all sorts of conditions applying to objects. It has been designed by **Shayne Flint** for navigating graphs, but it can also be used for many other object types.

ymuit

ymuit stands for *Yet More User Interface tools*. It groups tools used to implement the user interface of 3Worlds, mainly color palettes and management of graphic output, which can be useful for any javafx (https://wiki.openjdk.java.net/display/OpenJFX)-based interface.

tw-core

tw-core is the core of the 3Worlds software. It contains the base classes to design ecosystems and the simulator.

tw-apps

tw-apps contains the two applications needed to run 3Worlds, the *ModelMaker* and the *ModelRunner*.

tw-uifx

tw-uifx contains the javafx (https://wiki.openjdk.java.net/display/OpenJFX)-based interface classes for *ModelMaker* and *ModelRunner*.

tw-models

tw-models is a library of models designed with 3Worlds, including test and tutorial models.

tw-setup

tw-setup is used solely to create a jar containing all dependencies used by *ModelMaker* or *ModelRunner*. As such it is not strictly part of 3Worlds.

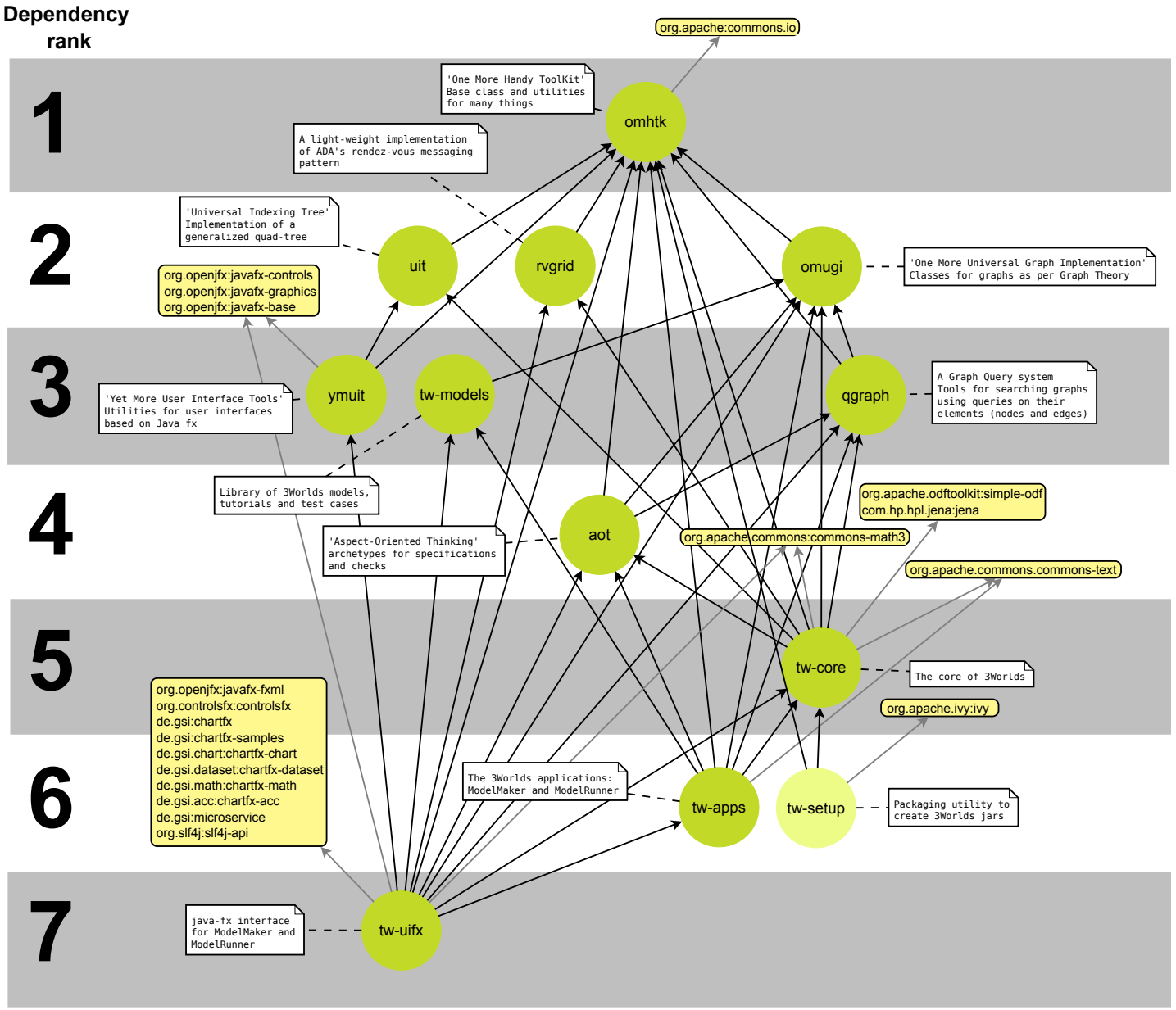

Figure 1. Dependencies of 3Worlds libraries. 3Worlds libraries are in green, third-party libraries are listed in the yellow boxes.

1.2. Library organisation

All 3Worlds libraries *must* adhere to the following directory structure:

```
<library_name>
    src             // the .java source files and only them
    test            // the test .java source files
    bin             // the .class binary files
    doc             // all the documentation-related files excluding javadoc
    lib             // downloaded .jar dependecies
    scripts         // dependency and versioning scripts
    javadoc         // generated javadoc files
    resources       // all non java files required to run the code
```

Each library is stored in a **git** repository. In other words, libraries are units of packaging for software distribution. However, not all the directories listed above are stored in the git repositories as they are generated by the IDE. These are:

- `bin`

- `lib`

In addition, the following directories are optional:

- `resources`
- `test`

1.3. Development tools

- The code is written in java (https://www.java.com/) **11**.
- These instructions are written for the eclipse (https://www.eclipse.org/ide/) IDE (version 2019-06).
- Code testing uses JUnit (https://junit.org/junit5/).
- Application interfaces use javafx (https://wiki.openjdk.java.net/display/OpenJFX), extended with controlsFx (http://fxexperience.com/controlsfx/).
- The documentation is written with asciidoctor (https://asciidoctor.org/) editor for eclipse (https://marketplace.eclipse.org/content/asciidoctor-editor). Asciidoctor enables production of **html**, **pdf** and **man** pages from the same source. You may find the following editors useful: AscidocFX (https://asciidocfx.com/), AsciiDocLIVE (https://asciidoclive.com) and the asciidoctor editor found on the eclipse MarketPlace.
- Dependency management is based on ivy (http://ant.apache.org/ivy) and ant (https://ant.apache.org/).
- Library repositories are managed with git (https://git-scm.com/), with the requirement that *1 library = 1 git repository*.

The development tools required are:

- **git**
- **asciidoctor**
- **eclipse**, including
  - support for **ivy** and **ant** (but that should come as a default when installing eclipse)
  - the egit (https://marketplace.eclipse.org/content/egit-git-integration-eclipse) plugin for eclipse, found on the eclipse MarketPlace

2. Working on existing 3Worlds libraries

To work on an existing library (<*library*>):

1. Launch **eclipse**. Create a new workspace.
2. If the Welcome window opens (the default), disable the `Always show Welcome at start up` checkbox and close the Welcome window (this option causes problems with the editor behaviour in some versions of eclipse).
3. Make sure eclipse is configured to use java **11** or greater by default (check in the `Window > Preferences` dialog box the `Java > Compiler > Compiler Compliance level`).
4. Shift to the Git perspective. You should now have a left window called `Git repositories`.
5. In this window, click on `Clone a Git repository`. 3Worlds is currently hosted by two gitlab servers:
   - at Sorbonne Université (SU) (https://grouper.ent.upmc.fr/gitlab/threeWorlds) in Paris, France
   - at the Australian National University (ANU) (https://gitlab.anu.edu.au/ThreeWorlds) in Canberra, Australia
   - These central repositories are managed by Jacques Gignoux (jacques.gignoux@upmc.fr) and Ian Davies (ian.davies@anu.edu.au). Ask them for an access to one of these servers. In the `URI:` field, enter one of:

- `git@grouper.ent.upmc.fr:threeWorlds/<library>.git` for SU
- `git@gitlab.anu.edu.au:ThreeWorlds/<library>.git` for ANU

Do not use the git repository URI starting with `https://<gitrepo>;` as this will not use the ssh key authentication, and will ask you for your git user and password every time you want to access the remote git repository.

6. Click `Next`.
7. Click `Next` again. Enter the location for your local git repository. The default location is usually a *git* directory under the user home.

Do **not** put the git repository inside the eclipse workspace, as this will cause trouble later.

8. Click `Finish`. After the download completes, a new entry will appear in the `Git repository` window. If this fails, check your access to the remote repository.
9. Expand the git repo by clicking on the black arrow on its left. This displays a list of items found in this project. In this list, expand the *Working Tree* entry. This should display a list of two entries, *.git* and *<library>*.
10. Right-click on *<library>* and select `Import Projects…`. In the dialog box, click `Finish`. NOTE: in older versions of eclipse, the new project may not be properly recognized. Upgrade eclipse if this happens.
11. Shift to the java perspective: the project contained in the local git repository is now present in the Package Explorer window, usually with many error messages.

To remove the errors, add the following libraries to **eclipse**:

1. **JUnit**:
   - Right-click on the project and select `Build Path > Configure Build Path…`
   - In the dialog box, select the `Libraries` tab
   - In the `Libraries` tab, select *Classpath*. This should enable the buttons on the right of the dialog box
   - Click the `Add Libraries…` button
   - In the dialog box, select *Junit* and click `Next`
   - Select version 5 of JUnit and click `Finish`

     The JUnit library should now appear in the dialog box under the *Classpath* entry.
2. **Ivy**:
   - Without closing the dialog box, select *Classpath* again to enable the buttons on the right of the dialog box
   - Click the `Add Libraries…` button
   - In the dialog box, select *IvyDE Managed Dependencies* and click `Next`
   - In the top field, replace `ivy.xml` by `scripts/ivy.xml` and click `Finish`
   - Click `Apply and Close`

     This should remove errors in the *src* folder.

If errors remain, it means the *ivy* local cache does not contain required dependencies of this library. To solve this see Trouble shooting.

3. Developing a new library for 3Worlds

This section describes how to create a new 3Worlds library in **eclipse**. If you plan to work on *existing* 3Worlds code, see instructions in section Working on existing 3Worlds libraries.

Before proceeding, check you have a copy of *VersionManager.java* and *VersionSettings.java* that should accompany this document. If not, ask the main developers (jacques.gignoux@upmc.fr or ian.davies@anu.edu.au).

Once you have decided on a name for your library (<*library*> from hereon), proceed as follows:

3.1. java project

1. Launch **eclipse** and follow the first three steps in the section: Working on existing 3Worlds libraries.
2. Create a new Java Project:
   - In the top menu, select `File > New > Java Project`.
   - Enter a project name (<*library*>) and click `Finish`.

3.2. directory structure

In the project, create the directory structure for your library (NB: by default, eclipse should have already created a ***src*** source folder):

1. in the `Package Explorer` window (usually on the left), right-click on your project name and select `New > Source Folder`
2. in the opening dialog box, enter ***test*** and click Finish
3. repeat this step to create the source folder ***scripts***
4. then create *non-source* folders ***resources***, ***doc*** and ***javadoc*** but this time selecting `New > Folder` rather than `New > Source Folder`.

It is important to create the correct type of folders (***source*** or ***non-source***). You can delete and recreate folders if you make a mistake.

We recommend that you store this file in the *doc* folder for further consultation during the development of your library (for example under a sub-folder called *dev*, for *dev*elopment).

3.3. version and dependency management

To setup the version management:

1. Select folder *scripts*. Right-click on it to create a package (`File > New > Package`) and name it ***fr.cnrs.iees.versioning*** when prompted. Click `Finish`.
2. Now import the files ***VersionManager.java*** and ***VersionSettings.java*** into this folder.
   - Right-click on the *scripts/fr.cnrs.iees.versioning* package and select `Import…`
   - In the dialog box, expand the `General` entry and select `File System`
   - Click `Next`
   - Click the `Browse` button facing the `From Directory` label and field.
   - In the dialog box, select the **directory** where your files to import are located.

- The dialog box should now display the directory tree on the left and a list of files on the right. Select **only** *VersionManager.java* and *VersionSettings.java* from this list.
- Click `Finish`. *VersionManager.java* and *VersionSettings.java* should now appear under *scripts/fr.cnrs.iees.versioning* with no visible error.

3. *VersionManager.java* should **never** be changed. In a unix-derived OS, it is a good idea to set this file permissions to read-only in order to prevent accidental modification of this file.
4. To modify *VersionSettings.java* to match the details of your <*library*>:
   - Double-click on the file name. It should open in the eclipse java editor window.
   - Carefully read the instructions given in the javadoc comment of the class (if you're familiar with hieroglyphics).
   - Modify as instructed the following fields (at the top of the class code): `ORG`, `MODULE`, `STATUS`, `LICENSE`, `LICENSE_URL`, `DESCRIPTION`, and `DEPS` if required. In the `DEPS` field you can provide a list of dependencies for both 3rd party and 3Worlds libraries. If you have no dependencies leave this entry empty. **NOTHING ELSE** should to be changed in this file.

   It is important to take some time to properly edit this file, as these fields will be used for versioning the whole library.
5. You can now run `VersionManager.main()` with no argument on the command line. In eclipse:
   - in the `Package Explorer` window, right-click on the class, select `Run As > Java Application`.
   - The console window in eclipse should now display:

     ```
     Upgrading "<library>" from version 0.0.0 to version 0.0.1 (Y/n)?
     ```

     Enter 'Yes'. You get this message in the console:

     ```
     Project scripts regenerated - Do not forget to refresh your eclipse workspace before going on.
     ```

   - As suggested, refresh your project (F5 on the project name). You should now see three new files in the `Package Explorer`:

     scripts/fr.cnrs.iees.versioning/current_version.txt

     This file holds the current version of your library (0.0.1 in this case). **Do not** edit by hand: it is entirely managed by `VersionManager`.

     scripts/ivy.xml

     This file is the ivy script needed by eclipse to manage dependencies of your library on other libraries.

     scripts/build/xml

     This file is the ant script needed by eclipse to manage dependencies *and* enabling you to pack your library into a jar file with proper versioning information for distribution.

     Since these scripts are generated they should never be edited by hand (because edits would be lost at the next version upgrade).

1. For eclipse to know about your dependency management, you must now tell it where to find ivy scripts:
   - Right-click on the project <*library*> and select `Build Path > Configure Build Path…`
   - Select the `Libraries` tab
   - Select the `Classpath` line. This should activate the buttons on the right-hand side of the dialog box

- Click on the `Add Library…` button
- In the dialog box, select `IvyDE Managed Dependencies`. Click `Next`
- In the top field, replace `ivy.xml` by `scripts/ivy.xml` and click `Finish`
- Right click on the *build.xml* file and select `Run as > 2 Ant build…`. Check the *publishJar* task and click *run*.
- Select the <*library*> and press `F5` to refresh your project.

Your project should now be ready to use the dependencies as listed in its (generated) *ivy.xml* file. If you look at your project (<*library*>) in the Package Explorer, you will see a *lib* directory which contains all the downloaded dependency jars, if in fact, you did list some dependencies and other files. This is why *lib* should not be managed by **git**: it is generated by eclipse.

3.4. local git repository for a new library

As your library is new, nobody knows about it and you should create a new git repository from scratch. This will later be pushed upstream to a common (e.g. github (https://github.com/) or gitlab (https://about.gitlab.com/)) server for sharing the library.

Before proceeding, make sure your git *user.name* and *user.email* on your local system are as you will be known on your proposed git server. Check the current setting by typing in a terminal `git config --list`. You can then set your user name by typing

```
git config --global user.name "FirstName FamilyName"
```

and your email address by typing

```
git config --global user.email "FirstName.FamilyName@example.com"
```

1. In eclipse, shift to the Git perspective. You should now have a left window called `Git repositories`.
2. In this window, click on `Create a new local Git repository`. When prompted, enter a path where you want this repository to be located stay on your hard disk (<*git-repo*> from hereon) and click `Create`. You should now see a new entry in the `Git repository` window.

Do **not** put the git repository inside the eclipse workspace, as this will cause trouble later. Rather, use the default location (usually a *git* directory under the user home).

3. Now go back to the Java perspective. Right-click on your project, select `Team > Share Project…`. In the dialog box, select <*git-repo*> in the `Repository:` list and click `Finish`. Your project is now managed by git.
4. Files in the project now have a small question mark attached on their icon. This means they are not yet tracked by git. To track files:
   - Right-click on <***library***> and select `Team > Commit…`
   - The `Git staging` lists a number of files with 'unstaged changes' (you may need to scroll to see all these entries). Some versions of **eclipse** may differ slightly in appearance.
   - A number of files are listed from the *lib* directory. We first need to exclude this entire directory (The **eGit** UI does not allow this intuitively). Double click on `.gitignore - <library>`.
     - In the `Local:.gitignore` you will see `/bin/`. This was added automatically during the previous steps. Add a new line with `/lib/`.

    - Add another new line with `/.settings/`
    - Click the "X" to close this window and select `Save` when prompted. Files in the *lib* and *.settings* directories are now removed from the `Unstaged Changes` list. Below, `<ORG>` represents the string you added previously to the `VersionSettings.java` file.
    - *.classpath -<library>*
    - *.gitignore -<library>*
    - *.project -<library>*
    - *3w-projects-setup.adoc - <library>/doc/dev*
    - *build.xml - <library>/scripts*
    - *current-version.txt - <library>/scripts/<ORG>*
    - *ivy.xml - <library>/scripts*
    - *VersionManager.java - <library>/scripts/<ORG>*
  - There are two more files to remove from tracking - ***.classpath*** and ***.project***. Select these two files (Ctrl + Mouse for multiple selection). Right-click on them and select `Ignore`. It is important not to track, and therefore share, these files as they represent local eclipse settings. If you open *.gitignore* as before, you will see that these files have been added to the list of untracked files together with the above mentioned directories.
  - You can now move the remaining files to the staged list by clicking on the green double 'plus' sign at the top right of the window
  - Enter a commit message (e.g. "Initial commit") in the right panel and click `Commit`. Your files are now stored in your local git repository.

3.5. remote git repository for your new library

Before you can share your new library with others, you must create a git repository for it on a shared server. 3Worlds is currently hosted by two gitlab servers:

- at Sorbonne Université (https://grouper.ent.upmc.fr/gitlab/threeWorlds) in Paris, France
- at the Australian National University (https://gitlab.anu.edu.au/ThreeWorlds) in Canberra, Australia

To create a new git repository on one of these servers, contact either Jacques Gignoux (jacques.gignoux@upmc.fr) or Ian Davies (ian.davies@anu.edu.au) to obtain an account on these servers. Once you have an account you can either create a project from that account or create a project remotely and push it up stream to that account. Here we will do the former.

1. With your favorite web browser, log in to your account on the server and create a project, here called <***library***>.
2. You may want to switch off `pipeline` processing for the newly created project unless you are sure you want this facility.
3. Once the project is set up, copy its URL. This will be of the form `git@<host>:<account name>/<project name>`
4. Back in eclipse, right-click on <***library***> and select `Team > Push Branch 'master'`
5. The first time you do this, eclipse opens a dialog box to enter the remote git repository details:
    - Leave the `Remote name` as `origin`
    - In the `URI:` field, paste URI of your remote repository. This has the form as given above.
    - Click `Preview`

- Click `Preview` again
- Click `Push`
- Click `Close`

4. Versioning

By versioning here we mean generating and managing meaningful version numbers for your library for distribution. This is *completely independent* from git version management.

4.1. aim and strategy

Every 3Worlds library has a 3-number-separated-by-dots version identifier. The three values represent MAJOR, MINOR and BUILD numbers. Deciding when and which value to increment is somewhat subjective. However, to try and maintain some consistency we suggest the following:

1. **BUILD**: This number should be increased when a bug, or suite of related bugs, has been fixed and tested.
2. **MINOR**: This number should be increased: (i) when a large refactoring has taken place; (ii) when new functionality has been implemented and is still undergoing testing; and, (iii) when important changes in 3rd party dependencies flow through to significant changes in the code.
3. **MAJOR**: This number should be at 1 when software is first publicly distributed. Thereafter, this number should be increased only when very significant new functionality has been added, tested and been found stable (e.g. a new GUI or integration with OpenMole).

4.2. method / how to

> **To avoid version conflicts, discuss the version increment with colleagues and decide who is to be responsible for making the version change. This is critical, as version changes are difficult to undo, especially when pushed to a central server.**

Version numbers are incremented by running `VersionManager.main()` (in package *scripts/fr.cnrs.iees.versioning*) with the **appropriate command-line argument**:

a. **BUILD**: no argument;

b. **MINOR**: `-minor` argument (this will reset the BUILD number to 0); or

c. **MAJOR**: `-major` argument (this will reset the MINOR *and* BUILD numbers to 0).

To pass arguments on the command line, you must create a *Run Configuration* (Main menu: `Run > Run Configurations…` etc. cf. the eclipse documentation for how to create run configurations) and then execute it.

1. Once you are clear about how to use the appropriate argument, run the *VersionManager*. The console window in eclipse should now display:

   ```
   Upgrading "<library>" from version <M.m.b> to version <N.n.c> (Y/n)?
   ```

   Enter 'Y'. You then get this message in the console:

   ```
   Project scripts regenerated - Do not forget to refresh your eclipse workspace before going on.
   ```

2. As suggested, refresh your <***library***> (F5 on the project name).

You should now see two new files in the `Package Explorer`:

scripts/fr.cnrs.iees.versioning/ivy-<M.m.b>.xml

This is a copy of the former *ivy.xml*, suffixed with the previous version identifer, for archive.

scripts/fr.cnrs.iees.versioning/build-<M.m.b>.xml

This is a copy of the former *build.xml*, suffixed with the previous version identifer, for archive.

The files *scripts/ivy.xml*, *scripts/build.xml* and *scripts/fr.cnrs.iees.versioning/current-version.txt* have also been rewritten to match the new version identifer.

3. Right-click on *scripts/build.xml* and select `Run As > 2 Ant Build…` (the second entry, not the first). In the opening dialog box, select the `publishJar` task and click `Run`. Things should happen in the console window and hopefully terminate like this:

```
...
[ivy:publish]   published <library> to /home/gignoux/.ivy2/local/fr.ens.biologie/<library>/0.0.2
/jars/<library>.jar
[ivy:publish]   published ivy to /home/gignoux/.ivy2/local/fr.ens.biologie/<library>/0.0.2/ivys/ivy.xml
BUILD SUCCESSFUL
Total time: 667 milliseconds
```

If you look into your *ivy* cache (*.ivy2/local/* in your home directory), you should now have a new sub-directory with a new version number (e.g. 0.0.2 here):

```
fr.ens.biologie
  <library>
    0.0.1
      ivys
        ivy.xml
        ivy.xml.md5
        ivy.xml.sha1
      jars
        <library>.jar
        <library>.jar.md5
        <library>.jar.sha1
    0.0.2
      ivys
        ivy.xml
        ivy.xml.md5
        ivy.xml.sha1
      jars
        <library>.jar
        <library>.jar.md5
        <library>.jar.sha1
```

All versioning information in *scripts/fr.cnrs.iees.versioning* is stored in the git repository. *VersionManager* archives the former versioning information as *build.xml* and *ivy.xml* files suffixed with the version numbers.

However, what is *not* archived is the state of the code at the time of version update: the development will go on happily and the changes will fade in the mists of the past…! If we want to be able to go back to a former version, we need to tell git about this version. This is simply done by using the *tagging* (https://git-scm.com/book/en/v2/Git-Basics-Tagging) capacity of git.

So, **just after a version upgrade** as explained above, and **before doing anything else** (e.g. routinely editing code), you **must**:

1. Prepare the commit of the files created by the version upgrade (just after a version update, you have changed many

files in the *scripts* folder):

- Go to the `Git Staging window in the java perspective (if you don't find it, right-click on project name and select `team > Commit…`. This will open it)
- Add all the files appearing in *Unstaged Changes* to *Staged Changes* by clicking on the double green plus in the top right corner of the window
- Write a commit message (for example: *"upgrading to version <N.n.c>"*)
- Click `Commit` (**NOT** `Commit and Push…`)

2. Set a git tag on this commit:
   - In the Package Explorer window, right-click on the project name, select `Team > Advanced > Tag…`
   - In the opening dialog box, enter the new version number you have just upgraded to (<N.n.c>) in the `Tag name:` field
   - In the `Tag message:` field, enter some description of this version change. Something meaningful and useful, e.g. 'fixed bug #543458754' for a BUILD change, 'refactored Query system' for a MINOR change, or 'added parallel execution support' for a MAJOR change.
3. Push the change to the remote git repository:
   - Click `Create Tag and Start Push…`
   - In the opening dialog box, check that the proper tag is associated to the proper commit and click `Next`
   - Click `Finish`
   - Click `Close`

> Good coordination between developers is very important to ensure these versioning operations go smoothly. Git is very permissive about tagging in remote repository (by default, tags are not pushed, and they can be easily overwritten). So please be careful.

5. Managing dependencies

It is easy to manage the dependency between your library and other software with *ivy*: you just have to add the appropriate dependency details in your *ivy.xml* file. However, since we generated this file, you must actually do it in the *VersionSettings.java* class. This is just as easy: you add them into your `DEPS` static field (see the comments associated with this field in *VersionSettings.java*). It is a good idea to set the revision identifier to `"+"` so that your library always uses the last version of the software. If you want to be more specific on which version of the dependency to use, you may type, e.g. `"[0.3.1,)"`, which will be interpreted as 'any version above 0.3.1 (e.g. 0.3.5 or 1.0.1 will work, but 0.2.67 will not).

6. Licensing

All work on 3Worlds libraries is distributed as free software under the GNU General Public license version 3 (https://www.gnu.org/licenses/gpl-3.0.en.html) (GPL.3) license. Every source file of 3Worlds libraries must contain a header with a reference to the GPL.3 and the full text of the license must be present in the distributed package.

There is a tool called Releng which might be better: https://www.codejava.net/ides/eclipse/how-to-add-copyright-license-header-for-java-source-files-in-eclipse

Eclipse provides a convenient way to automatically insert license text at the top of each newly created file. This text will be project specific as the project name must be mentioned in the license along with the project authors and a project

desciption. Below is some text you can copy and paste into the code template facility of eclipse. If, for some reason, you cannot copy and paste the text below, the text is can be found in *license-gpl3.txt* supplied with this document. You will need to do this for each machine you use (i.e. at home, work and travelling). To add the license to your project, follow these steps:

1. Right-click on your <***library***> (project) in the *Package Explorer*
2. Select `Properties…`
3. In the left-hand column, select `Java Code Style > Code Templates`
4. Check `Enable project specific settings`. **This is very important because the license text is project specific.**
5. In the `Configure generated code and comments:` list, expand `Code` and highlight `New Java files`
6. Click `Edit` and paste the license text at the top of the text in the `Pattern:` field, leaving the default references to variable unchanged.
7. Edit the text within hash markers with the <***library***> name, description and author details.
8. Click `OK`
9. Click `Apply and Close`.

Check again that you have flagged `Enable project specific settings` before proceeding to create project source code.

```
/*******************************************************************************
 *  #LIB# - #SHORT_DESC#                                                       *
 *                                                                              *
 *  Copyright 2018: #AUTHOR1#, #AUTHOR2# & #AUTHOR3#                           *
 *       #EMAIL1#                                                              *
 *       #EMAIL2#                                                              *
 *       #EMAIL3#                                                              *
 *                                                                             *
 *  #LIB# is #DESCRIPTION#                                                     *
 *                                                                             *
 *******************************************************************************
 *  This file is part of #LIB# (#SHORT_DESC#).                                 *
 *                                                                             *
 *  #LIB# is free software: you can redistribute it and/or modify              *
 *  it under the terms of the GNU General Public License as published by       *
 *  the Free Software Foundation, either version 3 of the License, or          *
 *  (at your option) any later version.                                        *
 *                                                                             *
 *  #LIB# is distributed in the hope that it will be useful,                   *
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of             *
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the              *
 *  GNU General Public License for more details.                               *
 *                                                                             *
 *  You should have received a copy of the GNU General Public License          *
 *  along with UIT.  If not, see <https://www.gnu.org/licenses/gpl.html>.      *
 *                                                                             *
 *******************************************************************************/
```

7. Writing test code

In a project the size of 3Worlds it is very important that all the code is tested. We use the standard JUnit (https://junit.org/junit5/) library for this purpose.

All test code (even the non-JUnit, custom code) must be placed in the *test* directory to keep production code packages clean.

Only non-abstract classes can be tested

1. To create a JUnit test case for class <*class*>:
2. In the Package Explorer window, right-click on <*class*> and select `New > JUnit Test Case`
3. In the opening dialog box:
   - tick the `New JUnit Jupiter test` check box
   - specify *test* as the source folder (it normally defaults to *src*)
   - click `Next`
4. Select the methods you want to create a test for and click `Finish`. The new test class should now appear in the *test* directory *under the same package name* as <*class*> in the *src* directory. This is important as it enables testing of protected methods (i.e., it is really the *same* package although the source directory is different for <*class*> and its test case).

If the new class appears with an error such as `The import org.junit.jupiter cannot be resolved`, it is probably due to the project not being told to depend on JUnit. This is easily solved:

1. In the Package Explorer, right-click on the project name, select `Build Path > Add Libraries…`
2. From the list, select `JUnit`, click `Next`.
3. Set the JUnit version to 5, click `Finish`
4. Click `Apply and Close`. After a few seconds, the errors attached to the *test* directory should vanish.

8. Generating the javadoc

Javadoc that is no more than a restatement of the code itself (e.g. documenting the return type of a method) is frustrating to find when you are looking for explanations. Insight cannot be automated - at least not by Javadoc. To be useful, javadoc should:

- state the *objective* of an item (typically a class or method): why was it written?
- explain its *use(s)*: often classes and methods are designed with a particular use in mind, and it is not always obvious to others what you meant. Misuses can be source of major flaws and code corruption.
- do not forget the *big picture*: what is the pattern or the strategy behind that code? This helps avoid misues.

Remember the great loneliness of the code developer in the face of 'The Algorithm', and see the javadoc as a way to share with your fellow developers some of the genuine intuitions you have transformed into astonishing code. Take some time to be clear. Put yourself in the shoes of the reader.

This said, generating a javadoc using eclipse is particularly easy:

1. Select the project. In the main menu, select `Project > Generate Javadoc…`
2. In the opening dialog box, verify that the check box `Use standard doclet` is ticked.
3. In the `Destination:` field, replace */doc* by */javadoc* to comply with the 3Worlds library directory structure described in Library organisation.

4. Click `Finish`. in the opening dialog box, confirm (`Yes to All`). You should now see a huge bunch of files appearing in the *javadoc* directory. Open *index.html* in a browser to explore the javadoc.

9. Writing documentation

A good software is useless without a good documentation (Confucius, The Book of Changes, 552 B.P.).

We recommend using asciidoctor (https://asciidoctor.org/) to produce it. Asciidoctor uses a 'lightweight markup' language to produce different document formats from a single source. It can produce html (https://www.w3.org/html/), pdf (https://acrobat.adobe.com/us/en/acrobat/about-adobe-pdf.html), odf (http://docs.oasis-open.org/office/v1.2/OpenDocument-v1.2.html) or man page (https://en.wikipedia.org/wiki/Man_page) documentation; in article, book or other custom format. Having said that, Asciidoctor is still not a mature environment, so expect limitations. In particular, some of the editing tools are not mature and conversion to other formats is a long way from perfect. Html seems good but pdf does not appear to be of professional quality. Its great benefit is that it can reference external files and thus more easily maintain consistency between code and documentation.

To install Asciidoctor, (it requires ruby (https://www.ruby-lang.org/)) and also the eclipse asciidoctor editor for maximal comfort. You also need to install asciidoc (http://asciidoc.org/) because Asciidoctor is an extension of Asciidoc.

Using the asciidoctor editor integrated in eclise is easy: you just have to select your asciidoctor file (a text file with extension *.adoc*) in the `Package Explorer` window, right-click on it and select `Open With > AsciiDoctor Editor`. The next time you open this file, you just have to double-click on it as eclipse will keep the association between that file and that editor in its preferences. The editor has a double window, one with the text and one with its compiled html output. Apparently, some little bug makes the output take a long time to show up the first time you open the file. Editing the file and saving it will cause it to run properly.

The downside of the ascii doctor plugin is that eclipse cannot do word-wrap. You can use carriage returns to get around this but it's less than satifactory and awkward if you also use an editor such as AsciidocFX or AscidocLIVE that do manage word-wrap.

Producing the exact doc files you want must be done outside eclipse in a terminal window, invoking `asciidoctor` on the command line. Type `man asciidoctor` in a terminal window to see the details of the syntax. In short:

- `asciidoctor <doc>.adoc` will produce a standard html documentation file named <doc>.html. To specify a custom output file name, use `asciidoctor -o <another-name>.html <doc>.adoc`. Option `-v` will tell you about errors in the source file.
- `asciidoctor -b docbook <doc>.adoc | pandoc -t odt -o <doc>.odt` will generate a (very crude) open office document. You need to install pandoc (https://pandoc.org/) to do this.
- Conversion to pdf usually requires an intermediate *docbook* format file:
  - `asciidoctor -b docbook -o <doc>.xml <doc>.adoc` will produce a docbook5 document.
  - `a2x -f pdf <doc>.xml` will convert it to pdf. `a2x` is part of asciidoc.

There is also a standalone Asciidoc editor (https://asciidocfx.com/). It nicely integrates the asciidoc(tor) tool chain, but the GUI is shaky and tends to crash unpredictably.

Finally there is asciidocLIVE (https://asciidoclive.com), an online method of editing. This site saves edits to your browser download directory in incrementally numbered files. Therefore, you will need to copy the most recent file from your brower download directory to you project at the end of an editing session.

10. Managing GitLab repositories

To create a new git repo for a library:

1. In the Menu bar, click on `Groups`. In the opening page, click on *threeWorlds*. This opens a page showing all the git repos / libraries existing in this group.
2. Click on the green `New Project` button. In the opening page, type the relevant project name, select the relevant visibility options and click on `Create project`.
3. This displays a page with all the information needed to use the new repo.

To delete a git repo:

1. In the Menu bar, click on `Groups`. In the opening page, click on *threeWorlds*. This opens a page showing all the git repos / libraries existing in this group.
2. In the left panel, click on `Settings` and select `Projects`. This opens a page where you can remove the projects. If you do not see the `Settings` button, ask the gitlab administrators to get the proper permissions on projects of this group. Usually, you can only delete projects that you have created yourself.

11. Trouble shooting

11.1. Interactions between ivy and eclipse

In principle, eclipse is able to manage dependencies based on the *ivy* script. However, there are sometimes problems emerging at version upgrades of dependencies. If you experience problems (e.g., `Class not found` error messages for a class belonging to a library you have declared in your dependency list), you may try one of these solutions:

1. In the Package Explorer window, right-click on `Ivy scripts.ivy.xml` and select `Clean all caches`. This erases the *cache* directory in the *~/.ivy2* directory, but not the *local* directory where the dependencies on local libraries (those of your projects) reside.
2. In the Package Explorer window, right-click on `Ivy scripts.ivy.xml` and select `Remove Ivy dependency management…`

If errors persist, then it may be worth doing a complete clean of the dependencies:

1. Go to your *.ivy2* repository, delete everything (i.e. *cache* and *local*)
2. Remove *ivy* from your project libraries (through `Project > Properties > Java build Path`). This should in principle (there seem to be display bugs - or maybe you have to use Refresh all the time) remove the `Ivy scripts/ivy.xml` entry from your Package Explorer.
3. Reconstruct all your local dependencies (starting from the top of the tree and following its branches in order) by running the *build.xml publishJar* task as explained in Versioning
4. Reload the *ivy.xml* dependencies by right-clicking on *ivy.xml* and selecting the `Add Ivy Library…` entry in the pop-up menu. This should reconstuct the proper list of jars under the `Ivy scripts/ivy.xml` entry in your Package Explorer.

As of 19/9/2019, this is the current solution we use when eclipse enters a cycle of meaningless compile errors:

1. synchronize all projects with file system (F5, refresh)
2. wipe out ivy cache (delete local and cache directories) NB this may be rather extreme - it is simpler to only delete the 3worlds libraries
3. delete all content in lib directories of all projects

4. regenerate all jars, THEN run ivy>refresh on every project, all this in dependency order
5. delete tw-dep.jar, ModelMaker.jar and ModelRunner.jar
6. regenerate them with TwSetup (few minutes)

11.2. Ant

1. Don’t run more than one Ant task at a time.
2. Eclipse site: http://www.apache.org/dist/ant/ivyde/updatesite

11.3. AsciidocFX

1. no word wrap in ascii doc plugin

Version 1.6
Last updated 2021-04-19 11:20:01 +0200