HMI JavaProjects guide

August 2012

A short overview for downloading, building, and running HMI projects

1

QuickStart

We assume that the following software is installed: `Java`, `ant`, and some `svn` client. If something is missing here, see chapter 3 on software requirements.

1. Allocate some directory for your Java projects. It will act as the base directory where directories for shared repositories will reside. It is also a convenient "root directory" for your own projects. For the rest of this document we will call it "`<JavaProjects>`". Example: For XP or Win-dows 7, `<JavaProjects>` would typically be a directory like `C:\JavaProjects`.

2. Use your favorite `svn` client to checkout the `HmiShared` directory from our HMI SVN repositories. It is found in the `http://hmisvn.ewi.utwente.nl/hmishared` repository.
   You should end up with the `HmiShared` directory as subdirectory of `<JavaProjects>`. Within `HmiShared`, you will find our common `ant` build scripts. A second important part of `HmiShared` is a `repository` directory containing library files, mostly `jar` files, that are used in most projects.

3. To check that it all "works" you might want to run one of the example projects, like our `HmiGraphicsDemo`:

   - Go to a dos-prompt in the root of the (example) project.
   - First run `ant resolve` This will retrieve the necessary library files from various repository directories like `HmiShared/repository` or `Shared/repository`, and puts these library files into your project's `lib` directory.
   - Finally run `ant`. This will compile, then run the main class of the project.

4. You might then check out some selected directories from the `Shared` directory. This repository contains, for example, data files for several specialized avatars and sound files for the Elckerlyc project. It is available from the same HMI SVN repository mentioned above.

5. It is advisable to check out the `HmiDemo` modules, because they contain several useful code examples for many purposes. The `HmiDemo` module is found in the `http://hmisvn.ewi.utwente.nl/hmidemo` SVN repository.

6. Now you can create a new Java project, or you can study one of the "demo" projects. After checking out, you might have a directory structure that looks like this:

   ```
   <JavaProjects>/HmiShared
   <JavaProjects>/Shared
   <JavaProjects>/HmiDemo
   <JavaProjects>/ExampleStudentProject
   <JavaProjects>/MyOwnNewProject
   ```

   ——-checkout Project non-recursively.

7. You can create a project, for instance by copying material from the `ExampleStudentProject`. demo. The `build.xml` file in the root of your newly created/copied project should be ok "as is". On the other hand, the `build.properties` file might need some editing. For instance, if your project needs library files from other projects, you can specify this in `build.properties`. See section **??** below for details.

8. When you have created a new project yourselves, you can share it with others by uploading it to the `http://hmisvn.ewi.utwente.nl/hmistudent` SVN repository. (See section **??** for more detail how to do this.)

2

Hmi Java Projects

This chapter describes in more detail how to deal with the Hmi Java Projects system.

2.1 Philosophy

It is often necessary to (re)build Java projects, even by persons who didn't develop that project. Since many different tools are being used, we have to agree on a number of basic conventions and procedures.

- We do *not* assume that everyone will be using the same development toolkit or any toolkit at all. But we *do* require that a project created and developed, say, by means of Netbeans or Eclipse, can be (re)build without any of these tools available.
- Since we have to use some common build tool, we have agreed to use *ant* as that "common" minimal platform.
- In principle every project lives in its own directory, containing all relevant source code, test code, project specific data, library files needed by that project, project documentation, etcetera. We have a preferred standard layout for the directory structure inside a project. A project also contains an `ant` buildfile (`build.xml`) and usually also a `build.properties` file. We require that we can (re)build and run a project using `ant`, without any reliance on development tools that might have been used to develop the project. To be clear: we don't want to install JBuilder or Netbeans or Eclipse or whatever just to build and run your project. The `ant` file can be either a simple "standalone" build file, but the preferred way is to use a very small build file that just links to our shared build file. (See below)
- We have a limited number of shared projects, all of which are available from our GIT repositories:
  - There is a "project" called `hmibuild`. It contains the shared `ant` build files. You must have this project in order to use our build system.

  - Shared software is available as source code or as compiled jar files. Most projects use the precompiled jar files, which are kept in a project's `lib` directory. (For tools like Eclipse or Netbeans you must do some configuring in order to use these library files, see below). We use a tool called `ivy`, used by our build files, for easy version management of `lib` files that relies on our web repository. (`hmirepo.ewi.utwente.nl`). Basically, when you type "`ant resolve`", then `ivy` will copy the library files needed for your project into the `lib` directory of your project. What will be copied is derived from a project file called "`ivy.xml`".
  - There are a few "projects", like `HmiResource`, that contains just "resource" data of all sorts, that is shared between projects. For instance, BML scripts, data for 3D scenes and avatars etcetera lives here. Usually, you can obtain such data also from the web repository, in packaged jar format. Sometimes, you want to actually see and modify that data, and in that case you will need to check out the relevant resource data from the git repositories.

- Projects can build "on top" of other projects, including external projects, in a hierarchical way. Projects that become *mutually or circularly dependent* should be refactored, for instance by merging mutually dependent projects together into a single project.

- Projects *import and export* class code and data in the form of `jar` files. There is no sharing of *source* code. This ensures that every project can be built stand alone, after importing the necessary library files.

- Import and export of class code and data is effectuated via shared repositories. On your local system, this is just some directory shared by various projects. Parts of these shared directories are also mirrored on our shared SVN repositories.

2.2 Ivy dependencies

Why dependency management

The contents of lib directories consists of `jar` files and/or "`dll`" or "`so`" files that are necessary for compiling and running the project. The basic strategy is that interdependencies between projects are via import/export of library `jar` files, in preference over direct source code dependencies. Of course we have to face the problem of project *versioning*. We have stable *release* versions of projects, but also less stable *beta* and *alpha* versions. here we have some conventions and rules. For instance, we do not want a stable release version of project X to be dependent on an unstable alpha version of project Y. The other way around, so an alpha version of X dependent on a release version of Y is OK of course. It will be clear that manual version management is not a good

idea: lot's of work, and error prone. Instead, we use a dependency manager, Ivy.

How does it work

Every project has an "`ivy.xml`" file that describes project dependencies. The `ant` build files, relying on the Ivy system, use these so called "ivy's" for *resolving* the contents of the project's `lib` directory. The system is based upon the notion of *configurations*, *versions*, and *status of versions*. For example, our build system uses "alpha", "beta", and "release" as possible status of some `jar` version. The ordering of these statuses is relevant. For instance, when you ask for a "latest beta" version of some module Ivy will choose the latest amongst published beta and release (but not alpha) versions of that module. Similarly, if you ask for a "latest alpha", anything is acceptable, but if you ask for a "latest release", then only versions classified as "release" are taken into consideration. Ivy allows for several "configurations" of the project, each with its own set of dependencies. Currently, we have configurations for producing alpha, beta, and release versions of the project itself, and a "test" configuration for (extra) dependencies needed for running tests. (For technical reasons we have two more configurations, called "master" and "default", which are discussed below) The work flow is roughly as follows: first move the project into either the alpha, beta or release configuration, second do your development, including testing etc, third publish an alpha, beta or normal release based upon the current configuration. When you actually publish, we attach a version number to the jar file. Also, the Ivy system records metadata concerning published modules based upon version numbers, to be used later on, when resolving for other projects. It does so by publishing not just a `jar`" file (or other "artefacts", as they are called by Ivy), but also an accompanying `ivy.xml` file, derived from the project's ivy at the moment of publication. In this way, the Ivy system knows not only about *direct* dependencies, but is also capable of resolving *recursive* dependencies. This means that, for instance, when my module X declares (just) a dependency on the `HmiGraphics` package, the resolve process will look into the dependencies of `HmiGraphics`. The result is that project X will receive `jar` files for `HmiGraphics`, but also for `HmiAnimation`, `HmiXml`, `HmiMath`, and `HmiUtil`, because of (recursive) dependencies. When some `jar` file is required more than once, say via a direct dependency and also via some indirect dependency, and the versions required are not consistent, then Ivy delivers the "latest" version. So, for instance, in the example above, if project X declares (direct) dependencies on the alpha version of `HmiGraphics` and the release version of `HmiXml`, while the alpha version of `HmiGraphics` declares itself a dependency on the beta version of `HmiXml`, then project X will receive that beta version of `HmiXml`. That should be ok: since we are asking for some alpha version (of `HmiGraphics`, we should allow other alpha and/or beta versions if `HmiGraphics` actually needs them, even if our own project would be satisfied with the most recent release version.

Version numbering

We use some *conventions*; the Ivy system does not require some particular numbering scheme, but uses version numbers to determine which version is "more recent". This includes a few subtle cases. For instance, Ivy knows that version `1.4-alpha` is before `1.4`, and that versions `2.0-rc1` and `2.0-dev384` are both *before* `2.0`. Our conventions:

- Basically, we use version numbers of the form `major.minor` where `major` are `minor` are numbers. Examples: `0.1`, `1.4`, `2.0`, `2.0.1`.
- We allow suffixes of the form `-alpha`, to denote alpha release versions, and `-rci` or `-devi`, where `i` some number, to denote "release candidates" and "developer versions". The latter are beta releases, intended to become a full release somewhere in the future. So, for instance, `2.0-dev384` is a developer version, working towards full release `2.0`.
- The `major.minor` form is the preferred form for both betas and full releases.
- The `major.minor-alpha` form is the preferred form for alpha releases.
- The `major.minor-rci` form can be used as a "release candidate" for release `major.minor`.
- The `major.minor-devi` form is used as a "developer version" for release `major.minor`. Such versions are produced during our "nightly" build process. (Should be running on a daily basis; but when testing during the nightly build produces error, no new versions are published that night)
- Version numbers for full releases and beta releases are to be unique. Once published, you cannot reuse the same version number for a new release anymore. For alpha versions this would be counterproductive, since alpha versions can be produced in rapid succession, with only very minor differences. For this reason, our strategy is that alpha releases will not be put on the repository, so are not shared, therefore do not need unique version numbers.
- Sometimes we use version numbers of the form `major.minor.maintenance`. This form is intended for releases that are bug fixes for normal `major.minor` releases. Say currently we have a release version `2.3`, and we already produced newer beta versions `2.4` and `2.5`. Then we detect an annoying bug in `2.3`. Now we could correct this and then produce a full release `2.6`, but that would also include our new/experimental/buggy beta and alpha code from `2.4`, `2.5`. The better solution here would be to use `git` to temporarily revert to the code of version `2.3`, do the bug fixing, then publish that as *maintenance* release version `2.3.1`. Note that this is a *release* version, not an alpha or beta, so it that will become the preferred one when resolving for a "latest release" version.

Ant targets for producing releases

The "current" version number is kept within the project's `manifest.mf` file. It is automatically updated by some of our `ant` targets, and used for Ivy publishing.

- `ant release` increments the minor version number, then produces a version with "release" status. When the current version happens to be a release candidate or developer version, then "incrementing the minor" actually means "stripping off" the `-devi` or `-rci` suffix.
- `ant majorrelease` increments the major version number, resetting the minor number to 0, then produces a release version. As before, when the current version happens to an developer or release candidate, then is replaced by "stripping off" the `-devi` or `-rci` suffix. The released version gets "release" status.
- `ant maintenancerelease` appends a maintenance number,or increments it when the current version already has a maintenance number, then produces a "release" version. Note that this does not take care of the process of "reverting" to the desired release version, nor does it (re)set the current manifest version number to the desired one. Currently this has to be done manually.
- `ant minorreleasecandidate` moves to the next minor version, appends the `-rc1` suffix, and produces a beta release. When the current version is already a "release candidate, the "rc" number is increased.
- `ant majorreleasecandidate` is similar, but first increments the major number, as usual for a major release.

2.3 Extended example

We describe an example of how to build versions of a number of dependent packages. We use here the HMI packages as an example.

- We assume that we start with existing source code, but no repository.
- For `HmiUtil`:
  - `ant resolve` This will resolve the project dependencies. For `HmiUtil` this is just a dependency on the JUnit framework, for testing. The `ivy` file for `HmiUtil` contains the line:

    ```
    <dependency org="junit" name="junit" conf="test" rev="latest.release"/>
    ```

    This line denotes that the `HmiUtil` project depends on the `junit` module from organization `junit`. We want the "latest release".
  - `ant`
  - `ant test`

2.4 Project directory layout

Java projects have a specific directory layout. It is possible to deviate, for instance, when you have a build tool that enforces a different layout. But we don't recommend this, especially because we want to have a structure that is easy to understand also for *others*.

Global setup and shared directories

Projects reside inside their own directory. Build tools like Netbeans or Eclipse are able to "link" such projects, for instance by importing jar files from one project into the other, or by automatic rebuilding projects on which your current project depends. Our build scheme allows for similar project dependencies, as described above, but does not integrate at the level of source code. As a consequence, you can allocate your project directory wherever you want, but for the sake of sharing libraries, sharing data, or sharing build scripts, we need a few *shared* directories. "Shared" here means mostly sharing among your own projects; it's not a directory within a shared filesystem. Real sharing is done by means of shared git repositories. (See below). These shared directories are assumed to be located in a common base directory that we will call here the "`JavaProjects>`" directory.

You are free to chose the location for `<JavaProjects>`, for example, it could be `C:\JavaProjects` on some Windows7 system or `/user/youraccount/javaprogs` on some Linux system.

Within `<JavaProjects>` we (currently) have a few directories. The most important one is "`hmibuild`", as it contains the shared `ant` scripts.

You can also choose the location of individual project directories wherever you want, as long as you specify that location *relative to the* `JavaProjects>` base. Each project has an `ant` build file called `build.xml`, and an`built.properties` file located in that project's directory. The `build.xml` file is usually a clone of the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="MyApp"  default="run">
   <import file="../hmibuild/build.xml" />
</project>
```

This example assumes that your project is located next to your copy of `hmibuild'', so that the file path \verb`../hmibuild/build.xml" is actually ok.

Inside the project directory

In principle, we have a fixed directory layout within a project:

```
<JavaProjects>
   <YourProject>
```

```
build.xml
build.properties
ivy.xml
src
   <Java source files for project packages>
test
   <Java Source Files for (JUnit) tests>
   lib
      <jar files for testing, like junit-4.8.jar>
lib
   <library files, i.e. jar files, .dll or .so files etcetera>
resource
   <data files, like configuration files, icon images, etcetera>
docs
   <project documentation, including generated Javadoc files>
build
   <class files generated by compiling this project>
dist
   <jar files, containing class files and resources for this project>
```

Sharing by means of GIT repositories

Projects can be shared via one of our GIT repositories. The Hmi related packages, as well as other shared material are found on the `hmigit.ewi.utwente.nl/` repositories. Access to these repositories are available on request; normally, you don't want the source code anyway, and you should use the packaged jar files, either stable releases, or from the daily build, via the "`ant resolve`" method.

What should be present *on a GIT repository*?

- Source files, data files, documentation: yes.
- Binaries, and class files: no.
- Library files, i.e. dll's, jar files etcetera: no.
- Netbeans or Eclipse project directories or files: no.
- The `ant build.xml` file and `build.properties` file: yes.

Project building using ant

We assume here that you are using the commandline version of `ant`. (There are tools like Netbeans or UltraEdit that allow you to issue these `ant` command from within the tool.)

Tip: `ant` is by default rather "verbose", showing for instance all sub-targets as they are called from a main target. To get rid of this, you can add a system environment variable called `ANT_ARGS` with value `-quiet` to your system. (In Windows this is done via the "Advanced" System properties panel.)

- As always, `ant -p` prints all available targets.
- `ant resolve` tries to check the current contents of the project's `lib` directory with similar directories within the repository. If necessary, repository files are copied into the project's `lib` directory, either because they were missing, or because they were older than the repository version. Which files will be checked must be specified within the build.properties file; see below.
- `ant resolveresources` is very much like `ant resolve`, but compares and copies *resource files*, that is, the contents of the `resource` directory.
- Simply typing "`ant`" should compile if necessary, then run the (current) main class of the project. This class is specified inside `build.properties` by means of the `run.main.class` variable. You can change by running `ant main`. This shows the current main class, and asks for a new main class. Simply typing a return will retain the current main).
- `ant doc` produces JavaDoc documentation inside the `docs/modulename` directory, where modulename is specified within `build.properties`. The packages to be included in the javadoc arespecifies also in `build.properties`.
- `ant main` shows current main class, then asks for new main class. (Simply `<Return>` won't change it). (There is a similar target `ueSetmain`, used in combination with tools like UltraEdit)
- `ant doc` produces JavaDoc documentation within the project's `docs` directory, including a zipped version inside the `dist` directory.
- `ant clean` deletes the `build` directory, including all class files.
- `ant jar` produces a `jar` file containing clas files and resources inside the `dist` directory.
- `ant release`, and `ant majorrelease` create a new minor or major release. First a new jar is created, then this jar file is copied into the repository directory where the project publishes its releases. It does *not* upload anything to the repositories. See the discussion on repositories below.
- `ant rebuild` is a special purpose version of the standard `ant build` target. It relies on the `rebuild-list` property, specified in `build.properties`. The latter provides a list of project directories, of projects on which the current project depends, directly or indirectly.
- `ant test` runs (JUnit) test cases for the project as a whole.

The build.properties file

The `build.properties` file contains a number of lines of the form `ant-property=value`. Most of these properties are set inside the `ant` files to default values, but when you set them explicitly inside `built.properties`, then they will overrule the defaults. The minimal `built.properties` file would contain a line similar to: `shared.project.root=..`, and a line like: `run.main.class=package.main`.

- `shared.project.root`: the relative path to the `<JavaProjects>` directory. (This one is important; with an incorrect path nothing will work) Example: `shared.project.root=..`
- `run.main.class`: The name of the Java class that should be run as "main" class. Note that you must specify the fully qualified classname, that is, including the package name. Example: `run.main.class=hmi.util.Info`.
- `resource.path`: when set, it specifies a path (as always, relative to `<JavaProjects>`), to a directory that will be included on the Java classpath. Directories found on this path are suitable for loading resources, for instance using the `hmi.util.Resources` class, or via Java methods like `getResourceAsStream`. Example: `resource.path=${shared.project.root}/Shared/repository/Humanoids`
- `run.jvmargs`: (extra) arguments for `java`, while running the main class. Example: `run.jvmargs=-Xms128m -Xmx1024m`
- `run.library.path`: Usually the java library path contains just the project's `lib` directory, so for instance `dll` files should reside in that directory. When you want to include other places, you can specify this with the `run.library.path` property. Example: `run.library.path=lib;${env.windir}/system32`
- `javadoc.packages`: the packages to be included when running javadoc, via `ant doc`. Example: `javadoc.packages=hmi.graphics.*`
- `module.name`: The (base-)name for `jar` files for this project, and also used for the javadoc directory inside the project's `doc` directory. The actual version of a `jar` file is determined from the "Specification-Version" line inside the `manifest.mf` file.
- `dependencies` and `resources`: a list of repository modules for the project's `lib` directory or `resource` directory. These modules will be updated when running `ant resolve` (for the the "dependencies"), or when running `ant resolveresources` (for the "resources"). Note that you can sometimes refer to *external* resources via the `resource.path` property, rather than resolving them into your `resource` directory.
  - The line of the form `dependencies=<dep1>, <dep2>, .....,<dep-n>` specifies *library* dependencies, that is, files that live within the project's `lib` directory.

  - The line of the form `resources=<dep1>, <dep2>, .....,<dep-n>` specifies *resource* dependencies, that is, files that live within the project's `resource` directory.
  - A dependency (i.e. an entry of the dependencies line) refers to some directory within the repository. The contents of that directory will be compared and copied if necessary.
  - A dependency of the form `<organization>/<module.name>`, or of the form `<module.name>` for projects that don't specify an "organization", refers to the "*latest stable*" release build on the repository.
  - A dependency of the form `<organization>/<module.name>/<version>` or `<module.name>/<version>` refers to some *specified version* within the repository. The allowed "`<version>`" values are: `latest`, `alpha`, `beta`, or some version *number* like `2.1`, `-3.1.4` or `_2`.

  Example:
  `dependencies=Sun/jogl, Hmi/HmiGraphics/beta, Hmi/HmiMath/1.1`

- `repositories`: A list of directories that act as repository, for the purpose of `ant resolve` and `ant resolveresources`. The default repository list includes `HmiShared/repository` and `Shared/repository`, so usually it isn't necessary to specify this property. Example:
  `repositories=HmiShared/repository, /C:/Myrepository`

- `publish.repository`: The path to the repository directory where *this* project will publish its releases. The default place is `Shared/repository`. Example: `publish.repository=/C:/MyRepos`

- `organization`: When specified, organization will be a subdirectory within `publish.directory` where releases are published. Example: `organization=Hmi`

- `rebuild-list`: a list of project directories (relative to `<JavaProjects>`), used by the `ant rebuild` target.

Our `ant build.xml` files are in essence identical for all projects. Occasionally, you might have to change the line:
`<property name="shared.project.root" location=".."  />`
The line above specifies that our shared root directory "`<JavaProjects>`" is one level up from the project directory, which is correct for most projects. If your project directory is *not* a direct subdirectory of `<JavaProjects>`, you should adapt this line inside `build.xml` accordingly. Customization can be dome by editing the `build.properties` file, that we describe below. If this is not enough, you can create a `build-customize.xml` file inside your project directory, containing extra `ant` targets as you see fit. An example `build.properties` file:

2.5 Repositories

A "repository" is a collection of library files and data. We have such repositories in the form of local repository directories, as well as (remote) SVN repositories. The local directories can be seen as largely as "cached" copies of the corresponding SVN versions. For each project, we maintain a module inside a repository, where various releases are kept, each in its own subdirectory. For instance, the `HmiUtil` module contains the `jar` files for the `hmi.util` package. The "publish repository" for the `HmiUtil` project, specified inside its `build.properties` file, is `HmiShared/repository/HMI`. Therefore, the releases are maintained within subdirectories of `<JavaProjects>/HmiShared/repository/HMI/HmiUtil`. Inside that directory you will find subdirectories `alpha`, `beta`, `latest`, and a few more carrying explicit version numbers, like `1.0`. The idea is that "`latest`" always contains the latest (stable) release version. Inside the directory, you will typically find a `lib` directory with content that will be added or updated for an `ant resolve` action.

2.6 Versioning

Projects can produce releases in the form of packaged `jar` files, carrying some version number, like for instance `HmiUtil-1.1.jar`. This version number is also present within the jar's *manifest*, which is kept in the `manifest.mf` file within the project directory. A version number has the form `<major>.<minor>.<micro>`, (where the `<micro>` part is optional). There are no strict rules for dealing with version numbering, but the following guidelines are sensible:

1. When a new version is produced that does *not* introduce or modify existing functionality, the specification version remains the same, and only the implementation version string is updated. (We automatically use the current date and time for the implementation version string.)

2. When functionality is *added*, the minor version of the specification version is incremented.

3. For more substantial modifications, in particular removal of functionality, or a complete redesign, the major version is incremented (and the minor version goes to zero).

4. There is no clear role for the micro version number. Typically, one can use this for small changes in functionality, especially when new beta releases are made. They are also used when a "stable" release turns out to contain bugs, in which case improved versions get a new micro version.

5. The `manifest.mf` file also includes an "Implementation Version" line, which specifies the date and time of the release. Implementation versions should not be *compared*, i.e. a "later" implementation should not be considered "better", just "different". These implementation version lines are updated automatically when a new jar file is produced.

The micro version number of a project can be incremented by running `ant newversion`. The idea is to increment the micro version of beta releases often, and semi-automatic. Increments of the major or minor version should be done *manually*, by editing the `manifest.mf` file. Versions of the manifest inside a `jar` file can be queried at runtime, for instance, to check that some library `jar` file is not out-of-date. For this purpose, we often add a special `Info` class to every package. When this is specified as the "main" class, it will print the manifest information, at least when run from a `jar` file. This `Info` class contains methods for easy checking version compatibility. For example, `hmi.util.Info.requireVersion("1.1")` will show an error message when the older `HmiUtil-1.0.jar` version happens to be present in the `lib` directory. (If you copy the `Info` class, you should change just the *package* line inside the source)

3

Software Requirements

Our HMI projects are based on tools and languages including the following ones:

1. An **SVN client**, like Tortoise SVN or SmartSVN. There is even a commandline svn.
2. The **Java development kit**, also called the JDK.
   - There are many variations of Java on `java.sun.com`, but what you need is the Java Standard Edition, also called Java SE.
   - If you have some version installed already, check the version from the command line with

     ```
     java -version
     javac -version
     ```

     Both should report some 1.7.x version or later.
   - If `java` or `javac` does not work from the command line, ensure that Java is installed, and that the java/bin directory is present on your executable path.// For Windows-XP this is probably the `C:\Program Files\Java\jdk1.6.x\bin` directory, for Windows 7 64–bit it is typically `C:\Program Files (x86)\Java\jdk1.6.x`. Note: Even on 64–bit systems, you can still use the 32-bit version of Java.
3. You can compile and run with the standard `javac` and `java` commands, but it is easier to do this using the **ant** build tool. Our Hmi project style even requires an `ant` build file for all projects.
   - Check: From the command line you can verify that `ant` is installed with:

     `ant -version`

     It should say something like "Apache Ant version 1.8.2 " If it is not installed, or you have a version older than 1.8.2:
   - Download `ant` from `ant.apache.org`.

- "Installation" of `ant`: just unzip, and ensure that the ant/bin directory is on your executable path.
- For large (re)builds, especially of HmiDemo projects with lots of dependencies, the default `ant` settings do not suffice, and `ant` might crash because of memory problems. When you run `ant` from the (Windows) commandline, you run `ant.bat` from the `ant/bin` directory. That file reads an optional settings file called `antrc_pre.bat`. It searches for this file in several places, including in your HOME directory or your USERPROFILE (Like "C:/Documents and Settings/username"). The "out-of-memory" problems can be solved by creating this file with a line like:
  `set ANT_OPTS=-Xmx256M -XX:MaxPermSize=256M`.

4. `ant` is good for basic building, and in combination with a good editor like **UltraEdit**, it can act as a basic development system. A good alternative is to use some development tool like **Eclipse** or **Netbeans**.

5. **Netbeans** is a comprehensive development tool, freely available from `www.netbeans.org`. The simplest option is to download the "complete" version, and to install only those modules you need. There are more plugins on the web, for instance. Netbeans is based on `ant` build scripts, which can be very convenient, but also causes some problems for projects that need to cooperate with others not using Netbeans. See the chapter on Netbeans.

6. **Eclipse** Popular alternative alternative for Netbeans.

7. **UltraEdit** is a good general purpose editor. EWI has this software available. (It is *not* free). With proper configuration, it can be used as a "mini" development tool.

3.1 Build file structure and implementation notes

Note: this section is intended for those who want to maintain or modify the internals of the build system.

directory structure

The relocation of files and directories is defined by means of `ant` properties. As usual, you can redefine these, or you can set their value, for instance in a build.properties file. Many of the property settings rely on other properties, indicating in the table below by means of the usual `ant` syntax, where `${propertyname}` denotes the value of `propertyname`.

relative.shared.ant.dir	HmiShared/ant
shared.project.root	Shared root dir
hmi.shared.project.dir.name	HmiShared
hmi.shared.project.dir	${shared.project.root}/${hmi.shared.project.dir.name}
relative.shared.ant.dir	HmiShared/ant
shared.ant.dir	${shared.project.root}/${relative.shared.ant.dir}
shared.template.dir	${shared.ant.dir}/template

3.2 Netbeans tips

We discuss some hints for Netbeans, in particular for version 7.2.

Integrating NetBeans with our build system

NetBeans has, in the background, an `ant` based build system that is not unlike our own build system. You can combine these, but you need to take some precautions when you create a NetBeans project. Potential problem: by default, NetBeans want to create its own build.xml `ant` file in the project directory. The solution is first create a project, including the appropriate build file, and the appropriate directory structure including the `lib`, the `src`, `test\lib`, and `test\src` directories. Check that it compiles, etcetera, but after that: do an `ant clean`. (Otherwise, NetBeans will complain that there is already a build directory in your project.)

Then there are basically two options to proceed: the first is to create a NetBeans "free form" project. In that case, Netbeans will simply pick up your existing `build.xml` file, and use that for it own builds. Disadvantage: it's ok for basic editing, compiling, etcetera, but you cannot profit from all Netbeans features. The second option is to create a new NetBeans "Java project from existing sources". In that case, NetBeans has its own build file, now called `nbbuild.xml`. Advantage: you can profit from all netBeans features, and it will not disturb the shared build files. Disadvantage: you must setup the NetBeans project carefully, as we will describe now.

1. We assume that you are using NetBeans 7.2 (or higher)
2. create, outside NetBeans, a basic project in line with our build system. If you have an existing project, that's ok as well. Ensure that you have "resolved" the project dependencies, so that the contents of the `lib` and `test\lib` directories is what you need. (Use `ant resolve`)
3. Ensure that the your project has no `build` directory: for instance use `ant clean`.
4. Now create a new project within NetBeans. Choose "New Java Project with Existing Sources".
5. Choose the correct project name (by default we pick the project directory name), and select the correct project folder. Leave the "Use dedicated

Folder for Storing Libraries" enabled, but choose a *different* name than ".\lib". For instance, choose "nbproject\lib". NetBeans stores some of its "own" info in its "lib" folder. Unfortunately, our version manager Ivy will discard such info from .\lib when you do an "ant resolve" operation, so we must give the NetBeans lib a special place.

6. "Next", add the src folder to the list of "Source Package Folders", and add the test\src folder to "Test Package Folders".

7. Now you can "Finish", and you have a NetBeans project. Except that it won't compile, because you must correct the library settings:

8. The NetBeans "Projects" panel on the left show two sets of libraries: "Libraries" and "Test Libraries"

   - Right click on "Libraries" and choose "Add JAR/Folder". Navigate to the project's lib folder and add all jars that you find there. Also add the project's resource directory here. (This will ensure that this directory is on Java's classpath, so the "getResourceAsStream" method for loading resource data will work) Leave the "Reference as Relative Path" option enabled: this prevents NetBeans from copying into its own nbproject\lib directory. Do the same with "Test Libraries", this time using the jars from test\lib, and the test\resource directory. As you can see we now have a junit-4.8.2.jar, that is present in our lib directory, but also two more NetBeans "libraries", one for JUnit3.8.2 and one for JUnit4.8.2. The latter two should be *removed*.

9. By now, the project should be in shape: no unresolved libraries, it compiles and runs, and JUnits test work.

10. Note that whenever you change the contents of the lib directories, you will have to go back to the project properties page (from the File menu), and reestablish the correct libraries settings. For example, say you had some file "guava-r06.jar" inside your lib directory, but after the resolve operation it has been replaced by "guava-r07.jar" NetBeans now automatically removes guava-r06.jar from its libraries, but does not auto-understand that it needs guava-r07.jar instead. Right-click on the Netbeans project name, and choose "Resolve Reference Problems...". Select guava-r06.jar and click "Resolve". Now you can navigate to the project's lib directory, and select the new guava-r07.jar. For the time being, NetBeans is now satisfied. In reality, it has not really removed the reference to guava-r06.jar, but it has added a redirection inside your own "private" netbeans settings. So next time that you resolve, it cannot find guava-r06.jar again. You can correct this by first removing the guava jar from the NetBeans library, then adding the correct version. problem: you *cannot* remove it while the reference problem is not solved. In the end, the easy solution is to first remove *all* jar files from lib (and

also from `test\lib` if you expect changes over there) before the resolve, then add them back after the resolve.

building jar files

- By default, compiled class files are included in the jar file produced by a build. But also *all* files inside the src directory, except for .java and .form files are included. That is convenient if you want to include "resource" files, like data files, images , icons etcetera: put them in the appropriate source directory, and use the Java getResourceAsStream() method for reading the data. Although this works, our strong preference is to include "resources" into its own `resource` directory within your project. (Our `ant` build system puts your project's own `resource` directory on the classpath, as well as other directories that you specify in the `build.properties` file) Netbeans auto-include of src documents is also not so nice if you have some secret-remarks.txt, or some huge remarks.doc word file in your source directories. In this case, you must set the files to *exclude* for packaging. For instance, via:
  `Build | Set Main Project Configuration | Customize...`,
  or via right-clicking the project tab, and selecting "Properties". Choose the "Packaging" tab, and to the "Exclude from jar file", for instance, add `,**/*.txt, **/*.doc`.

subversion

- First *checkout of an existing project*: Use the "versioning", and select the checkout options (for svn). The repository name depends on the project, but will typically be something like "http://hmisvn.ewi.utwente.nl/hmi", where the last "hmi" identifies the specific repository from all svn repositories managed by Hmi. Use your own (EWI) username and password. After clicking "next" you can *browse* the repository. Select the folder you want to have. This will typically be a folder containing a complete "project" like, for instance, the Hmi/HmiAnimation folder. The commandline svn allows for specifying the local directory name; in Netbeans, you can only select the local directory where the folder will be placed. In the example, if you select a local directory called "javaprojects", then a subdirectory called " HmiAnimation" (i.e. without the HMI part) will be placed in that directory.

- By default, netbeans wants to put everything on a subversion repository except for the private parts of the nbproject directory. We don't want the nbproject directory at all, since it sharing it doesn't work too well, and is the cause for a lot of annoyances.