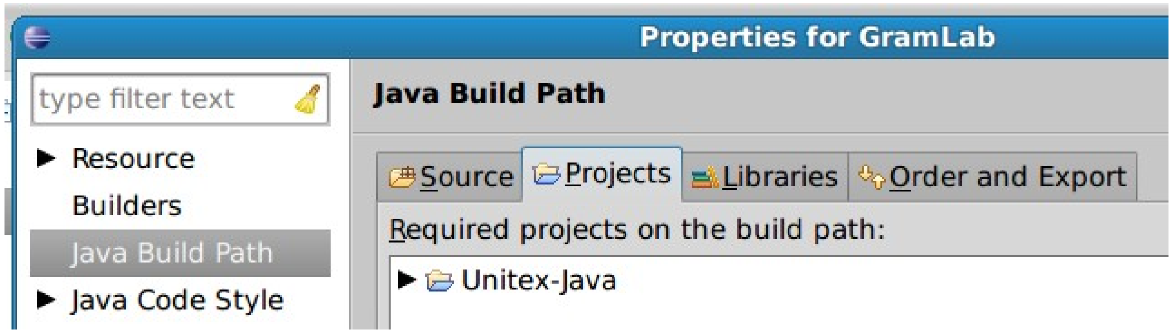
An overview of Unitex/IDELing Java code

Sébastien Paumier

LIGM, Université Paris-Est

Technical details

- use Java 1.6 and Swing
- both Unitex and IDELing have been developed with Eclipse
- you must set Unitex as a required project for IDELing:



Properties for GramLab

type filter text

- Resource
- Builders
- Java Build Path
- Java Code Style

Java Build Path

Source | Projects | Libraries | Order and Export

Required projects on the build path:

- Unitex-Java

Unitex logic

- user vs system directories
- a directory per language
  - you can't have different settings for two tasks on the same language
- language-specific hard-coded constraints (semitic mode, char-by-char mode, etc)
- you can work with one language at a time

IDELing logic

- a workspace containing projects
- you can have several projects opened at the same time
- similar projects can use common things with the dependency system
- you can configure everything
- main goal: fixing all tiny annoying details from Unitex

The challenge

- how to reuse as much code as possible from Unitex without breaking the previous logics ?
- `Gramlab.jar` uses `Unitex.jar` as a library
- introduction of an abstraction layer in Unitex code so that IDELing can override some configuration things

ConfigModel

- this interface lists methods needed for obtaining configuration information like:

```
public File getAlphabet(String language);
```

- `language` has the following meaning:
  - Unitex: name of the current language directory
  - IDELing: name of the concerned project; `null` means the current project

ConfigManager

- to access to the actual information, you have to ask to the `ConfigManager`:

```
ConfigManager.getManager().getAlphabet("biniou");
```

- in Unitex, an instance of `ConfigManager` is used
- in IDELing, an instance of `ProjectPreferences` is used

Configuration storage

- in Unitex:
  - a file named `Config` in the language directory
  - produced by the an instance of `Preferences`
  - some things are hard-coded

Configuration storage

- in IDELing, there are 4 files:
  - `pom.xml`: maven configuration file
    - `Pom.java`
  - `project.local_config`: user's private preferences (text editor, last graphs used...)
    - `ProjectLocalConfig.java`
  - `project.preferences`: Unitex preferences (font, ...)
    - `Preferences.java`
  - `project.versionable_config`: project settings to be shared on SVN (preprocessing config, ...)
    - `ProjectVersionableConfig.java`
- top-level object: `Project.java` that delegates to the previous classes

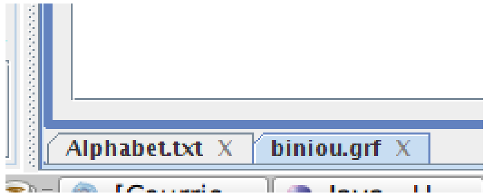
Configuration storage

- in IDELing, the rule is to save configuration files on every modification:

```
public void validateAndSave() {
    if (validateConfiguration(project,false)) {
        try {
            project.saveConfigurationFiles(false);
        } catch (IOException e) {
            JOptionPane.showMessageDialog(null,
               "Error while saving your project configuration:\n\n"
                +e.getCause(), "Error", JOptionPane.ERROR_MESSAGE);
        }
    }
}
```

Frames

- `InternalFrameManager`: allows each project to have its own `JDesktopPane` in IDELing
- `FrameFactory` objects to manage frames
- `TabbableInternalFrame`: used to provide a tab access to frames in IDELing



- `KeyedInternalFrame`: identify frames with a special value (often a `File`)

Launching commands

- `Launcher`: launches command sets, with or without console logging
- `MultiCommands`=list of `AbstractMethodCommand` objects that can be:
  - Unitex programs: `DicoCommand`, etc
  - other external programs: `MvnCommand`, `SvnCommand`, etc
  - method calls: `CpCommand`, `MkdirCommand`, etc

Launching commands

- `ProcessInfoFrame`: runs commands and displays their outputs into a frame
- you can run commands without this frame:
  - `ExecParameters`: allows you to control what to do with process output and error streams
  - you can use it invoking directly `Executor.start()`

Adding a new command

- create the `XxxCommand` class with methods to setup the arguments
- make sure to use properly typed arguments and not evil things like:

```
public XxxCommand input(String file) {
...
}
```

- if it is a command used by Unitex, add it in `HelpOnCommandFrame` to make it visible in the help frame

Big files

- support for large text files and HTML concordance files:
  - `BigTextArea`, `BigTextList`
  - `BigConcordance`, `BigConcordanceDiff`
- involves file mapping
- because of java bug #4715154, requires the phantom reference trick as in `TextAsListModel.reset()`

SVN support

- `svnktclient.jar`: a standalone 1.7 SVN client with only one `.svn` directory
- invoked from `SvnCommand`
- `SvnExecutor`:
  - error message processing with `SvnCommandResult`
  - `getSvnInfos`: for each file, creates a `SvnInfo` object describing the file status; used to display information in the tree

SVN support

- `SvnExecutor.getSvnStatusInfo` returns a `SvnStatusInfo` instance that lists:
  - unversioned files
  - added files
  - modified files
  - removed files
  - files in conflict
- used to refresh the tree and to prepare commits

SVN credentials

- for every svn operation, first try without credentials
- on failure (the `SvpOpResult value` is `AUTHENTICATION_REQUIRED`), we try again with `SvnCommand.auth`
- credentials are stored by the SVN client in `$HOME/.subversion/auth`

Ignore/add policy

- by default, ignore `..*` `*.fst2` `*.bin` `*.inf` `target` `dep` `build` `project.local_config` `diff`
  - could be overriden by a manual `svn add`, but you don't really want that
- `.grf` files are forced to be considered as binary files in order to avoid `svn diff3` merging them as text
- don't add any file above the `src` directory, except gramlab configuration files

$HOME/.gramlab

- global configuration file listing:
  - known SVN repositories
  - current workspace
  - current project in current workspace
  - other opened projects in current workspace

```
svn_repositories: 2
http://foosvn.univ-mlv.fr/svn/test/fr
http://my.other.svn.server.com/svn/biniou
/home/paumier/my_gramlab
en
fr
```

Maven support

- `PomIO` is responsible for I/O on `pom.xml` files
- for each project, a `Pom` object describes the GAV and the dependencies, if any
- `MvnCommand` is used invoke `mvn` as an external program:
  - under Windows, we launch `cmd /c mvn` because one cannot not really launch a `.bat` file from a JVM

Maven support

- we test if the two required gramlab artifacts are installed
- if not, we install them:
  - `App/assembly/pom.xml`: pom used to package projects as `.zip` artifacts
  - `App/pom.xml`: gramlab parent pom
- see `Pom.getXXXCommand` methods

Packaging a project

- we generate a ant task in the pom file that is responsible to copy and/or compile files to be packaged
- as this task may invoke UnitexToolLogger in a portable way, the maven command has to be invoked with its path as an argument:

```
mvn -Dunitextoollogger=<path to it> ...
```

Getting dependencies

- the command `mvn dependency:unpack-dependencies` places dependencies in the `dep` directory
- `dep` is made read-only in order to prevent users to try editing files in it
- it must be made writeable again before modifying the project's dependencies

Hornet nests

- graph display objects:
  - `GenericGraphicalZone`, `GraphicalZone`, `TfstGraphicalZone`
  - `GenericGraphBox`, `GraphBox`, `TfstGraphBox`
- IDELing workspace management:
  - `GramlabFrame`, `ProjectManager`, `fr.gramlab.workspace.*`
  - workspace tree refresh is a nightmare!