
	Titre du document : Livrable 4.6.2 – Livrable 4.6.3
	Référence : Livrable 4.6.2 – Livrable 4.6.3
	Version du 10/10/2011

## Livrables 4.6.2 et 4.6.3

*Export des exigences depuis un outil de modélisation UML*


<b>Livrable du au titre du projet</b>	COCLICO
<b>Lot</b>	
<b>Tache</b>	WP4- tâche 6
<b>Livrable</b>	L4.6.2 et L4.6.3

<b>Rédacteur(s)</b>	<b>Vérificateur(s)</b>	<b>Approbateur(s)</b>
Séverine Rambaud		

	Titre du document : Livrable 4.6.2 – Livrable 4.6.3
	Référence : Livrable 4.6.2 – Livrable 4.6.3
	Version du 10/10/2011


Documents applicables
Annexe technique au projet COCLICO

Documents de références (pour information)

	Titre du document : Livrable 4.6.2 – Livrable 4.6.3
	Référence : Livrable 4.6.2 – Livrable 4.6.3
	Version du 10/10/2011


## Gestion des versions

N° de version	Date	Auteurs	Modification apportées
1	10/10/11	Séverine Rambaud	Version initiale du document


	Titre du document : Livrable 4.6.2 – Livrable 4.6.3
	Référence : Livrable 4.6.2 – Livrable 4.6.3
	Version du 10/10/2011

## Sommaire

1	General Tool Presentation.....	6
2	Installation and launch.....	6
3	Application structure.....	7
4	Adding repository connectors.....	7
5	Accessing embedded help.....	7
6	Synchronizing.....	8
	a) Configuration options.....	8
	b) Configuration files.....	8
7	General application presentation.....	9
8	SCM.....	9
9	Maven modules.....	10
	a) requirements-modules.....	10
	b) requirements-repo-api.....	10
	c) requirements-synchronizer.....	10
	d) requirements-synchronizer-cli.....	10
	e) requirements-*-repo.....	10
10	Repository API, implementing a repository.....	11
	a) com.objetdirect.coclico.requirements.repository.api.Repository.....	11
	b) com.objetdirect.coclico.requirements.repository.RepositoryFactory.....	11
	c) Implementing a repository.....	12
11	Unit Testing.....	14
12	Status reporting.....	15
13	Embedded help.....	16

	Titre du document : Livrable 4.6.2 – Livrable 4.6.3
	Référence : Livrable 4.6.2 – Livrable 4.6.3
	Version du 10/10/2011

14 Synchronizer internals.....	17
a) Repository factory discovering.....	17
b) Repository management.....	17
c) Status reporting.....	17
d) Lifecycle.....	18
15 CLI internals.....	18
16 More documentation.....	18
17 Dedicated tracker and EA file.....	19
18 Glossary.....	19

	Titre du document : Livrable 4.6.2 – Livrable 4.6.3
	Référence : Livrable 4.6.2 – Livrable 4.6.3
	Version du 10/10/2011

# 1 General Tool Presentation

The synchronizer takes requirements from an input requirement repository and synchronizes them with one or more output repositories. The synchronization involves creating the new requirements and updating the ones that already exist in the repository.

There are supporting methods in the tool to read the requirements from any repository and accessing online help.

# 2 Installation and launch

Installation is done by unzipping the contents of requirements-synchronizer-cli-<version>.zip in a folder.

The synchronizer comes pre-packaged with:


- The command line tool
- A batch launcher for Windows
- The synchronization library
- A set of repository connectors
  - Enterprise Architect as an input repository
  - Codendi as an output repository
  - Novaforge Requirement as an output repository

In order to run the synchronizer, a Java 6 JRE must be installed on the computer.

To launch the synchronizer, simply run the following command, which will display the embedded help, from a command prompt:

```
synchronizer.bat
```

On non-windows systems, it is possible to adapt the batch script to launch the appropriate Java command

	Titre du document : Livrable 4.6.2 – Livrable 4.6.3
	Référence : Livrable 4.6.2 – Livrable 4.6.3
	Version du 10/10/2011

## 3 Application structure

The application is composed of:

- The synchronizer command (for Windows): synchronizer.bat
- The actual synchronizer and supporting libraries, packaged as JAR files in the “lib” folder. The contents of this folder may not be changed.
- The available repository connectors and supporting libraries, packaged in the “connectors” folder. The connectors shipped with the synchronizer have their dependencies in the “lib” folder, along with the synchronizer’s own libraries.


## 4 Adding repository connectors

To add a repository connector, drop its .jar file and dependencies in the “connectors” directory.

## 5 Accessing embedded help

The embedded help can be accessed by running the synchronizer without arguments or with the help command:

```
synchronizer.bat help
```

	Titre du document : Livrable 4.6.2 – Livrable 4.6.3
	Référence : Livrable 4.6.2 – Livrable 4.6.3
	Version du 10/10/2011

## 6 Synchronizing

Configuration and options for the synchronization and available connectors is vastly documented in the embedded help.

In order to synchronize, the synchronizer must be passed the command “synchronize” as the last parameter.

### a) Configuration options

Each repository activated with the --output or --input switches can be configured using custom options. The embedded help lists the options that the connectors can use.

Example: synchronizing Enterprise Architect with Novaforge

```
synchronizer.bat -i ea -o nova -D \
  nova.url=http://mycompany.com/novaforge/ -D \
  ea.file=C:\path\to\ea\file.eap synchronize
```

### b) Configuration files

Sets of pre-configured files can be written and used to configure the synchronizer. These files are activated using the --config switch and must reside in a readable location on the file system.


Example: synchronizing Enterprise Architect with Codendi using config files

```
codendi.config:
output=codendi42

codendi42.url=http://codendi.mycompany.com
codendi42.login=mySynchronizerLogin
codendi42.password=mySynchronizerPassword
# See embedded help to get those values for the project
codendi42.trackerId=1337
codendi42.groupId=666

ea.config
input=ea
ea.file= C:\path\to\ea\file.eap

Command:
synchronizer.bat -c codendi.config -c ea.config synchronize
```

	Titre du document : Livrable 4.6.2 – Livrable 4.6.3
	Référence : Livrable 4.6.2 – Livrable 4.6.3
	Version du 10/10/2011

## 7 General application presentation

The synchronizer takes requirements from an input requirement repository and synchronizes them with one or more output repositories. The synchronization involves creating the new requirements and updating the ones that already exist in the repository.

There are supporting methods in the tool to read the requirements from any repository and accessing online help.


The application is written in Java 6. It uses Spring for dependency injection and repository discovery; properties files for general configuration, embedded help and internationalization; and Maven for building, packaging and library management.

## 8 SCM

The SCM for the application is SVN. The repository is on ObjetDirect's Codendi forge:

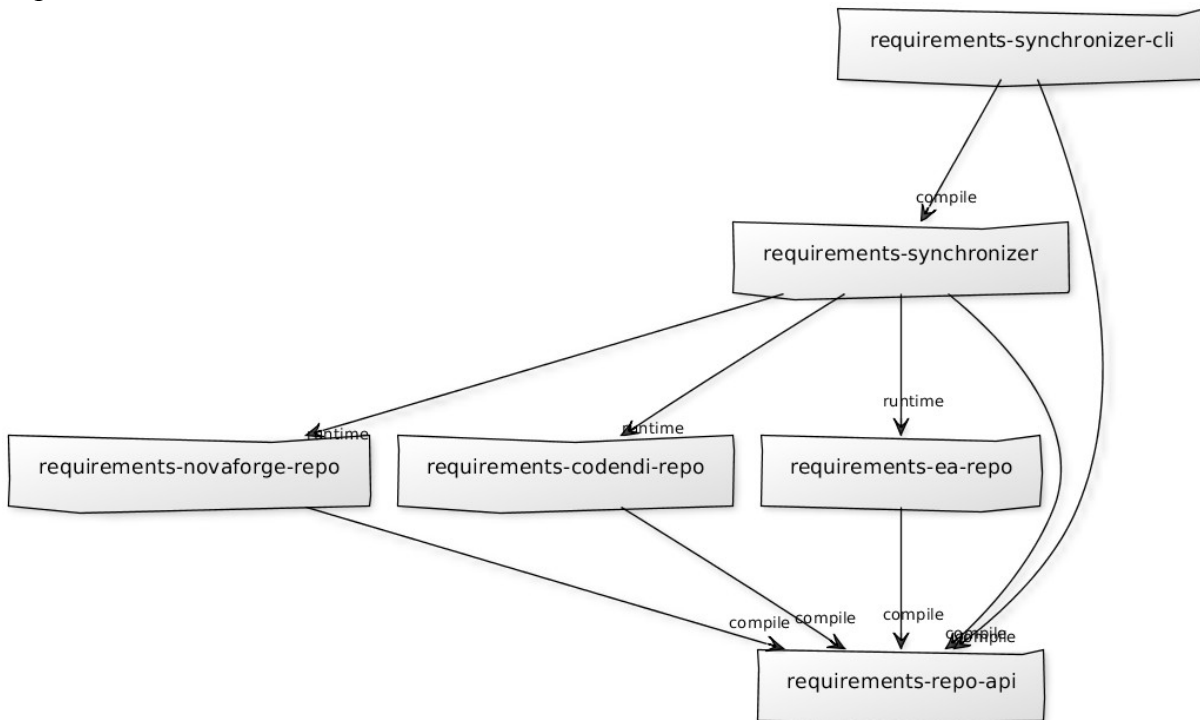
<https://partners.codendi.com/svnroot/codex/dev/branches/coclico/requirements/>

In the trunk there are several projects: one per Maven module, plus the Maven root project.

	Titre du document : Livrable 4.6.2 – Livrable 4.6.3
	Référence : Livrable 4.6.2 – Livrable 4.6.3
	Version du 10/10/2011

## 9 Maven modules

The project is split between APIs, synchronizer, user interfaces (CLI) and repositories implementation. All modules follow the Maven standard structure.



### a) requirements-modules

This is the maven root project, used to build the whole application as a Maven reactor.

### b) requirements-repo-api

This is the Repository API, plus standard test fixtures for repositories. All projects depend on it.

### c) requirements-synchronizer


This is the synchronizer itself, with supporting facilities and API (status reporting, repository discovery...). It has a runtime dependency on the packaged repository implementations.

### d) requirements-synchronizer-cli

This is the command-line interface to the synchronizer. It implements all the status reporting, documentation generation and synchronizer usage for use with a command line.

### e) requirements-\*-repo

These are the repository implementations.

	Titre du document : Livrable 4.6.2 – Livrable 4.6.3
	Référence : Livrable 4.6.2 – Livrable 4.6.3
	Version du 10/10/2011

## 10 Repository API, implementing a repository

The repository API defines a set of interfaces that must be either implemented or used by an implementer in order to have a repository usable with the synchronizer.

### a) `com.objetdirect.coclico.requirements.repository.api.Repository`

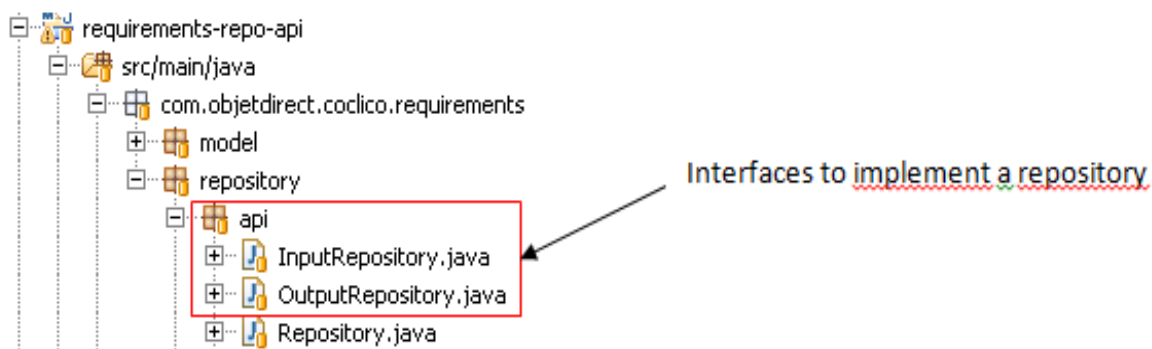
This is the repository interface itself. Implementation must take care of loading required libraries, opening connections, and converting standard requirements (defined by interface `com.objetdirect.coclico.requirements.model.Requirement`) into their own internal format.

The repository can expose configuration properties, but does not need to validate their values. It should not connect automatically, since this will be driven by its factory.

The repository must have a way to get a `StatusReporter` implementation in order to report its status (see Status reporting).

Implementations never implement the `Repository` interface directly. Instead, they implement either the `Input` or `Output` form of the repository:


`com.objetdirect.coclico.requirements.repository.api.InputRepository`  
`com.objetdirect.coclico.requirements.repository.api.OutputRepository`



### b) `com.objetdirect.coclico.requirements.repository.RepositoryFactory`

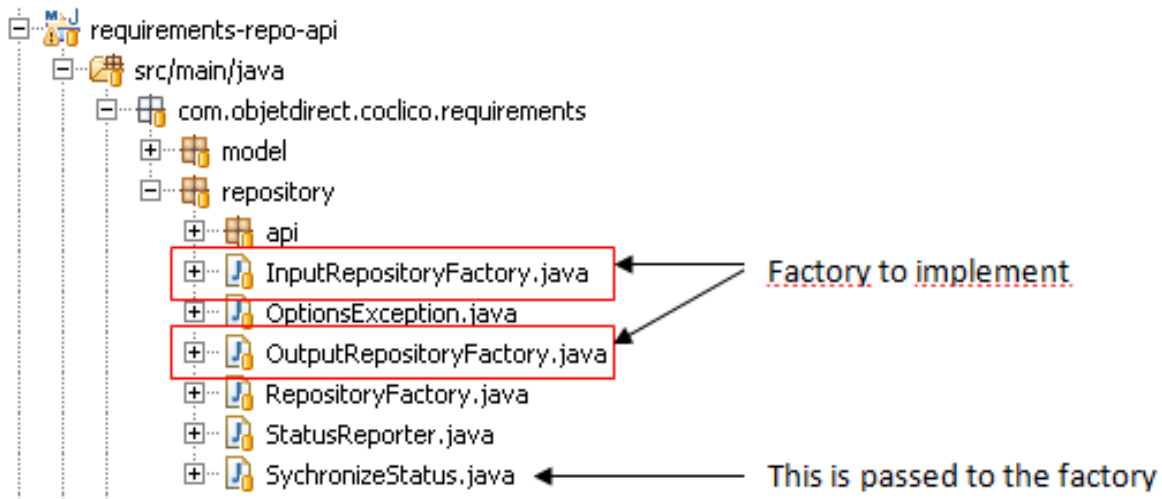
The repository factory takes care of the life cycle of the repository. Its tasks are:

- Communication with the synchronizer
  - Auto discovery
  - Status reporting
  - Embedded help
- Creating and managing a repository

	Titre du document : Livrable 4.6.2 – Livrable 4.6.3
	Référence : Livrable 4.6.2 – Livrable 4.6.3
	Version du 10/10/2011


- Creating the repository
- Configuring the repository from synchronizer options
- Obtaining and passing status reporting facilities
- Destroying the repository cleanly

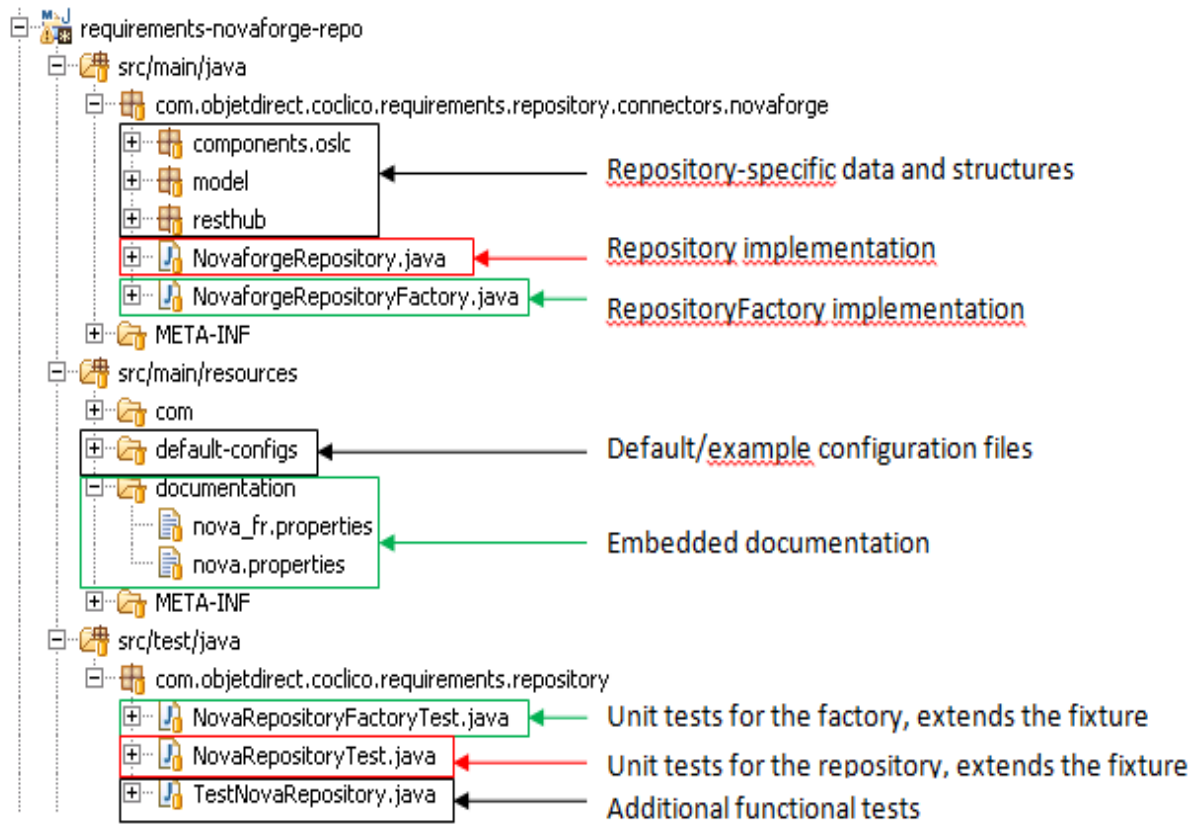
Factory implementation must be Named objects (annotated with `@Named` annotation) in order to be discovered by the synchronizer.




## c) Implementing a repository

- 1) Implement the repository connector. This step is completely custom depending on the type of the repository
- 2) Implement status reporting for the repository (see Status reporting)
- 3) Create the factory class, give it the `@Named` annotation
- 4) Implement repository options (see the Javadoc for `RepositoryFactory.getOptions()`)
- 5) Implement repository creation, including validating and setting options (see the Javadoc for `RepositoryFactory.createRepository()`)
- 6) Implement repository cleanup
- 7) Write embedded documentation for the repository and its options (see Embedded help)

	Titre du document : Livrable 4.6.2 – Livrable 4.6.3
	Référence : Livrable 4.6.2 – Livrable 4.6.3
	Version du 10/10/2011



	Titre du document : Livrable 4.6.2 – Livrable 4.6.3
	Référence : Livrable 4.6.2 – Livrable 4.6.3
	Version du 10/10/2011

# 11 Unit Testing

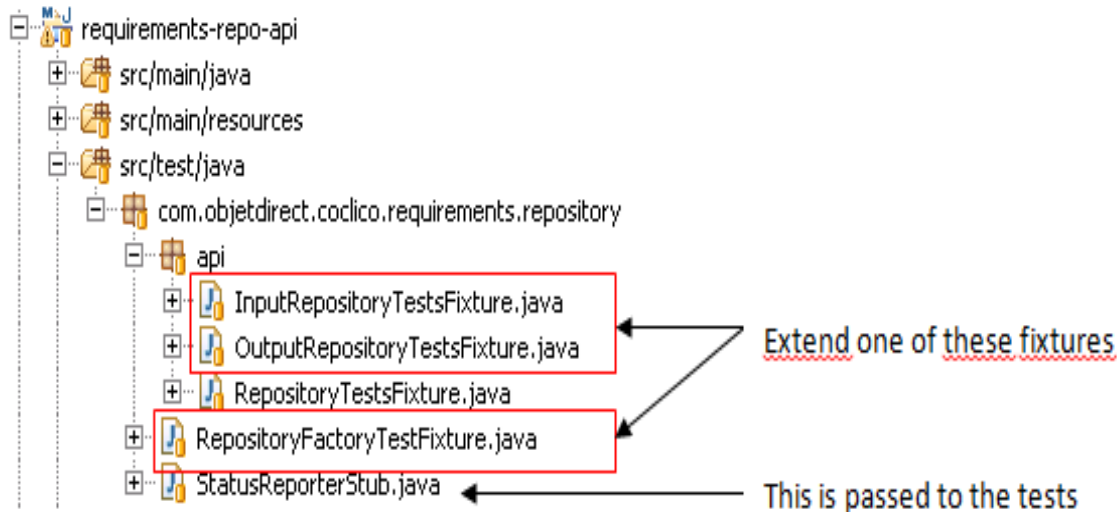
In order to unit test the factory and repository, test fixtures are provided that will check for correct creation of the repository and repository factory, option passing, and status reporting. The fixtures use the jUnit4 and Mockito frameworks. They are included in the build using this dependency:


```
<dependency>
  <groupId>com.objetdirect.coclico</groupId>
  <artifactId>requirements-repo-api</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <classifier>tests</classifier>
  <scope>test</scope>
</dependency>
```

To use them, just make your test class inherit the appropriate fixture and provide the required abstract methods:

```
com.objetdirect.coclico.requirements.repository.api.InputRepositoryTestsFixture
com.objetdirect.coclico.requirements.repository.api.OutputRepositoryTestsFixture
com.objetdirect.coclico.requirements.repository.RepositoryFactoryTestFixture
```

You should of course complete the tests with any relevant unit test for the specific repository.



	Titre du document : Livrable 4.6.2 – Livrable 4.6.3
	Référence : Livrable 4.6.2 – Livrable 4.6.3
	Version du 10/10/2011

## 12 Status reporting

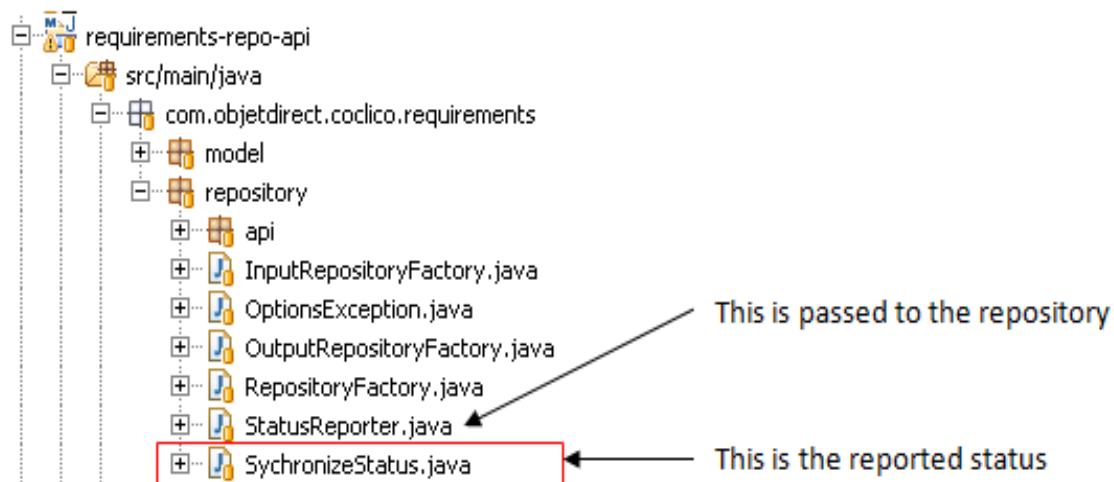
The synchronizer uses a mechanism called status reporting to retrieve information about the synchronization lifecycle and status, such as:


- Current phase of the synchronization
- Current requirement being processed
- Completion or failure of the process

A status reporter implementing `com.objetdirect.coclico.requirements.repository.StatusReporter` is created by the synchronizer. This reporter is passed to the factory, which must pass it to any repository it creates.

The only method in this interface takes a status in parameter. Predefined statuses are available in `com.objetdirect.coclico.requirements.repository.SynchronizeStatus`, but custom ones can be created and sent.

The current implementations of the status reporter are the CLI implementation and a stub implementation for testing.



	Titre du document : Livrable 4.6.2 – Livrable 4.6.3
	Référence : Livrable 4.6.2 – Livrable 4.6.3
	Version du 10/10/2011

## 13 Embedded help

The embedded help is generated by using information from the repository to get the repository description, possible options and descriptions for them.

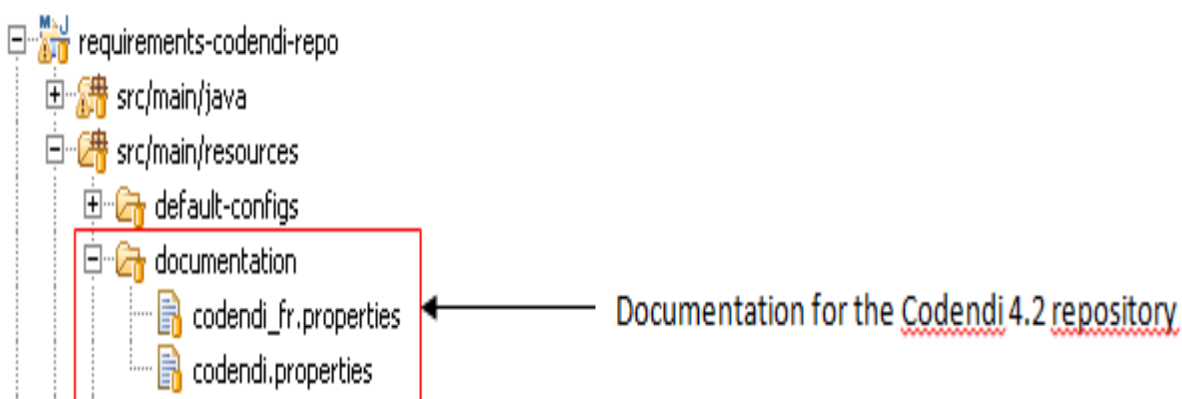
There are two main things about the help system:


- 1) The possible options are given by the `RepositoryFactory.getOptions()` method
- 2) The factory exposes a `ResourceBundle` from the `RepositoryFactory.getDocumentation()` method, that contains key matching the options for title and description.

There are several examples of this functionality usage in the provided repositories, and the Javadoc for the methods provides extensive information about the bundle keys. A few rules must be observed:

- The title string for the repository or an option must be short
- The description must inform about:
  - the format of the parameter
  - the fact that it is mandatory (if relevant)
  - the way to find the right value for an option (i.e. what part of a particular page URL must be used, etc.)
- The bundle files are found in `src/main/resources/documentation`
- The bundles can be internationalized using the default `ResourceBundle` mechanism
- The default language is American English

Embedded help can also be used to tweak the status reporting output a little. For example, the Novaforge repository overrides the reporting's "sides" strings to use HTTP transport terminology. This is interface-specific, so different interfaces could use different overrides.



	Titre du document : Livrable 4.6.2 – Livrable 4.6.3
	Référence : Livrable 4.6.2 – Livrable 4.6.3
	Version du 10/10/2011

```

codendi.properties  codendi_fr.properties
1title=Codendi 4.2
2description=Envoie les exigences à un tracker Codendi. Le tracker doit avoir une \
3forme prédéfinie (un template est inclus dans la documentation). Création et mise \
4à jour sont supportées.
5
6codendi42.url.title=URL de Codendi 4.2
7codendi42.url.description=URL de base de l'application web Codendi 4.2. Dans la \
8barre d'URL du navigateur, il s'agit de la partie avant le premier /. Exemple : \
9https://codendi.projet.objetdirect.com
10codendi42.login.title=Login
11codendi42.login.description=Nom d'utilisateur de l'utilisateur Codendi.
12codendi42.password.title=Mot de passe
13codendi42.password.description=Mot de passe de l'utilisateur Codendi.
14codendi42.groupId.title=ID du projet Codendi
15codendi42.groupId.description=GroupId du tracker d'exigences. Il s'agit de \
16l'ID projet de Codendi, disponible dans l'URL de la page des outils de suivi, \
17paramètre d'URL "group_id".
18codendi42.trackerId.title=ID de tracker Codendi
19codendi42.trackerId.description=TrackerId du tracker d'exigences. Il est \
20disponible dans l'URL de la page de rapport "Defaut" du tracker, paramètre \
21d'URL "tracker".

```

1 Codendi 4.2's documentation/codendi\_fr.properties (help in French)

## 14 Synchronizer internals

### a) Repository factory discovering

The synchronizer discovers repositories by injecting the lists of RepositoryFactory interfaces at startup. This is entirely done by dependency injection, which is why all factories must be @Named. The startup script for the synchronizer configures the class path so that any repository available will be discovered.

Once injected, the lists can be walked to find specific repositories or generate embedded help.

### b) Repository management

When a repository is configured for use, the synchronizer creates it from its factory through a managing proxy.

The reason why is that this proxy keeps track of the generic parameters for the factories to ensure that the right factory closes its own repositories. This also allows compilation without warnings.


The proxy also helps in ensuring that the created repositories are closed properly.

See the internal class FactoryHandler for more information.

### c) Status reporting

The status reporting interface used by the synchronizer is more complete than the one used by the repositories, since it also includes methods to report generic synchronizer lifecycle.

The exact reporting implementation is injected by the interface.

	Titre du document : Livrable 4.6.2 – Livrable 4.6.3
	Référence : Livrable 4.6.2 – Livrable 4.6.3
	Version du 10/10/2011

## d) Lifecycle

The lifecycle of a synchronization is as follow:

- 1) Report that we are going to read, get the repository reporter for this
- 2) Filter the execution options to get only those relevant to the repository
- 3) Get the factory for the repository
- 4) Obtain a repository from the factory, passing it reporter and options
- 5) Read the requirements
- 6) Do the same for each output repository, but write the requirements
- 7) Whatever happens, close any opened repositories

The lifecycle for a read is nearly the same, except there is no distinction on input/output repository, and only a read phase is done.

## 15 CLI internals

The command line uses Arg4J to parse command line arguments and generate arguments help. It also can do some option setting using properties files.

The command line and configuration file parsing logic is embedded in:

```
com.objetdirect.coclico.requirements.cli.CommandOptions
```

The actual command execution and embedded help creation is taken care of by:


```
com.objetdirect.coclico.requirements.cli.Main
```

Status reporting implementations are:

```
com.objetdirect.coclico.requirements.cli.CliSynchronizerStatusReporter
com.objetdirect.coclico.requirements.cli.CliReporter
```

## 16 More documentation

The classes and packages in the application are heavily documented through Javadoc. The documentation can be found directly in the files at the appropriate places.

	Titre du document : Livrable 4.6.2 – Livrable 4.6.3
	Référence : Livrable 4.6.2 – Livrable 4.6.3
	Version du 10/10/2011

## 17 Dedicated tracker and EA file

A sample EA file and a file containing the requirement tracker structure has been provided in the SVN repository (L4.6.3):

[https://partners.codendi.com/svnroot/codex/dev/branches/coclico/requirements/Tracker\\_Requirements.xml](https://partners.codendi.com/svnroot/codex/dev/branches/coclico/requirements/Tracker_Requirements.xml)  
<https://partners.codendi.com/svnroot/codex/dev/branches/coclico/requirements/requirements.eap>

The EA file is a sample of the requirements model to use.

The xml file containing the requirements tracker structure can be used to configure a Codendi project with a required requirements tracker.

## 18 Glossary

**Command:** The command the synchronizer executes, i.e. “help”, “read”, “synchronize”. The available commands are documented in the embedded help. The command is the last word on the command line, and defaults to “help” if no command is given.

**Connector:** Implementation of the client or library for a repository. It includes programs that can talk with the repository and link the repository to the synchronizer. Connectors are packaged in Java JAR files, and placed in the “connectors” folder of the application. They can have dependencies on supporting JAR files which may be placed in the “connectors” folder too.

**Embedded help:** Help displayed by the tool when ran with no command or with command “help”. It contains extensive documentation of the switches and commands, as well as an exhaustive list of the available repositories and their options with associated documentation.

**Input repository:** repository where requirements were filled in. It will be a source to the synchronization

**Repository:** place where requirements are stored or entered

**Repository option:** A configuration option for a specific repository. No two repositories take the same options, so they are documented as part of the embedded help.

**Output repository:** repository where requirements will be modified, used, addressed. This is often a forge. It will be the target of a synchronization.

**Synchronization:** Action of reading the requirements in an input repository and creating or updating them in one or several output repositories. Synchronization is the whole point of the synchronizer tool.

**Switch (command line):** Option passed on the command line to configure the synchronizer (e.g. -i, -o...). They often have two forms: a complete form starting with two dashes (e.g. --config) and a short form starting with one dash (e.g. -c).